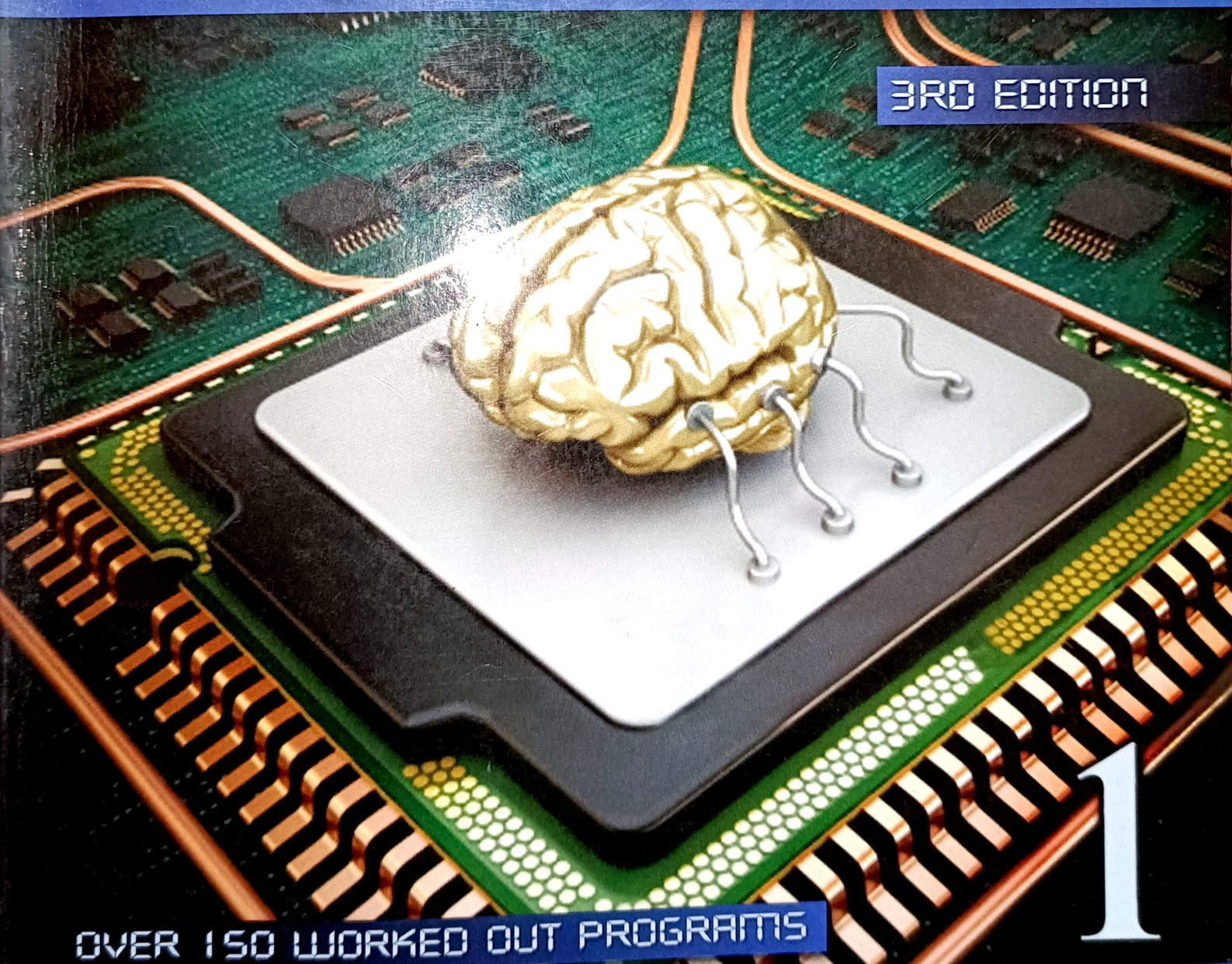
500+ MCQs

Rudiments of Computer Science

3RD EDITION



OVER 150 WORKED OUT PROGRAMS

1

ap ACADEMIC PUBLISHERS

J Bhattacharya

Rudiments of
COMPUTER SCIENCE

As per syllabus of West Bengal Council for Higher Secondary Education

[For Class XI]

Joyrup Bhattacharya
Dept. of Computer Science
South Point High School, Kolkata

ACADEMIC PUBLISHERS
5A Bhawani Dutta Lane, Kolkata-700073
E-mail : contact@academicpublishers.in
Website : www.academicpublishers.in

© Reserved by the author

First Published 2010
Second Edition 2013
Third Edition 2016
Reprint June, 2017
Revised Reprint 2018
Reprint 2019
Reprint 2020
Reprint 2022

ISBN : 978-93-83420-64-3

Price : Rupees Six hundred only.

All rights reserved by the author. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying or otherwise, without the prior permission of the copyright holder.

Published by B. K. Dhur of Academic Publishers, 5A, Bhawani Dutta Lane, Kolkata-700 073, laser type setting by Studio Michaelangelo, 5A, Bhawani Dutta Lane, Kolkata-700 073 and printed at Rajendra Offset & Graphics, 11 Panchanan Ghosh Lane, Kolkata-700 009.

This book is dedicated to
all those students
who have inspired me
to write this book

Preface to the Third Edition

The third edition of the book 'Rudiments of Computer Science' has been thoroughly revised and updated based on the bifurcated syllabus of Computer Science and the new question pattern introduced by the WBCHSE in recent years.

Some of the chapters on programming have been restructured and rewritten for a better understanding of the subject. To cater to the needs of the students, more worked out programs have been included in a separate section at the end of each chapter on programming. More than 500 multiple choice questions have been included in total and their answers given at the end of the book. Numerous short answer type and long answer type questions are also included at the end of each chapter. For the chapters on programming, more programming problems have been included at the end of each chapter to reinforce the learning experience.

I am grateful to various people who have helped me in taking out this book. First and foremost I am thankful to Sri Dipankar Dhar of Academic Publishers for giving me the time required to incorporate the updates before the publication of this book. I am thankful to my family for their support and also to South Point High School for granting me the permission to write and publish the book.

Thanking everyone associated with the publication of this book and hoping that the book will help both teachers in teaching the subject and students in understanding the same.

September 2, 2015

Joyrup Bhattacharya

Preface

The book 'Rudiments of Computer Science' has been written based on the new divided syllabus of Computer Science for the plus two level of the Higher Secondary course as introduced in 2007. I have been teaching the subject for almost a decade. After the revised curriculum for Computer Science (classes XI and XII combined) was introduced in 2001, there were no good books available that could fully cater to the needs of the students. Accordingly, they had to refer to several books to cover the complete syllabus. However, since most of the books followed were for the college level, students found them difficult to study. To help them overcome this problem I had prepared various study materials which had then helped the students immensely in understanding the subject.

Two years back, Sri Dipankar Dhar of Academic Publishers approached me to write a book on Computer Science. I thought of compiling and rewriting those class notes in the form of a book. The book has been written keeping in mind the difficulty faced by students in understanding certain topics and programming concepts. These topics have been covered accordingly with plenty of explanatory diagrams and examples. Moreover, the book has been printed in bi-colour (thanks to the publisher) to highlight the important sections and make the diagrams vivid and easy to understand. More than a hundred worked out C programs have also been included. A special section has also been included at the end of each chapter on C to illustrate various common programming mistakes. Hope these will help the students in getting a better understanding of the concepts.

To help locate different topics easily the book includes a full fledged general index, C- program index (both program-number wise and topic wise), C keyword and library function list, DOS and UNIX command list, acronym list, ASCII character list, list of various 'differences' covered in the book and other enhancements and value additions.

I am grateful to those who have helped me in writing this book. First and foremost I am thankful to Sri Dipankar Dhar for his immense patience as he had waited for more than two years to finally publish this book. I had never found him impatient or angry for submitting the chapters between long intervals! I am thankful to my colleague Sri S. K. Shee for giving me valuable suggestions for some chapters. I am also thankful to my students Anshuman Pal, Aritra Bose, Shrey Shubham, Asmit De, Mouktik Sarkar, and Arkaa Dev Roy who have sincerely carried out the proof-reading of the book. If the reader encounters any mistake or typographic error in the book, I request the reader to kindly inform the same at the publisher's email address so that the mistake can be rectified in the next edition. Other suggestions to improve the quality of the book will also be entertained. I am also grateful to Arjun Pakrashi for helping me in collecting various material for the book and to Mainak Basu for preparing the index for the book. I am also thankful to my wife for her undaunted support and finally I thank South Point School for granting me permission to write and publish this book.

Thanking everyone associated with the publication of this book and hoping that the book will cater to both teachers teaching the subject and students in understanding the same.

Doljatra
28 February, 2010

Joyrup Bhattacharya

Detailed Syllabus of Computer Science

CLASS XI

(Theory–70 Marks, Practical–30 Marks)

→ Important Facts.

→ Random Facts.

→ Definitions & Short Notes.

A. BRIEF REVIEW OF COMPUTER SYSTEMS : (30 MARKS)

(i) ***Evolution of Computers and Computer Organization :***

- **Evolution of Computers**
  - Abacus, Napier's Bone, Pascaline, The Babbage Machine
  - Stored Program Concept, Von Neumann Concept/Architecture
- **Computer Hardware Generations**
  - First, Second, Third, Fourth and Fifth Generation of Computers;
  - Components, Advantages, Disadvantages
- **Concept of Circuit Integration**
  - SSI, MSI, LSI, VLSI, ULSI
- **Classification of Computers**
  - Analogue, Digital, Hybrid Computers
  - Mainframe and Super Computer
  - Mini, Micro, Laptop Computer
- **Computers in Modern Society**
- **Concept of Data and Information, Data Processing**
- **Brief description of each functional block of a computer**
  - Block Diagram of a Computer System
  - Input Devices (Keyboard, Mouse, Scanner, Touch Screen, OMR, OCR, MICR, Graphic Tablet, Barcode Reader, Light Pen, Microphone, Joystick)
  - Output Devices
    - Monitor – CRT, LCD
    - Printer – Impact Printers (Dot Matrix Printer), Non-Impact Printers (Inkjet Printer, Laser Printer)
    - Plotter
  - Central Processing Unit : CU, ALU
  - Storage Devices
    - Primary Memory : RAM (DRAM, SRAM), ROM (PROM, EPROM, EEPROM, UVPROM)
    - Secondary Memory : Magnetic Media (HDD, FDD), Optical Media (CD, DVD, Blue-Ray Disk)
    - Cache Memory
    - Flash Memory
  - Communication Bus
    - System Bus – Address Bus, Data Bus, Control Bus, Power Bus

(ii) ***Data Representation :***

- **Number Systems**
  - Concept of Non-Positional Number System
    - Roman Number System
  - Concept of Positional Number System
    - Decimal, Binary, Octal and Hexadecimal Number System
  - Conversion
    - Inter-conversion between Decimal, Binary, Octal and Hexadecimal Numbers (Whole numbers and Fractions, using Double Add and Half Add Methods)
  - Arithmetic
    - Addition, Subtraction – Decimal, Binary, Octal and Hexadecimal Numbers
    - Multiplication, Division – Binary Number System only
  - Different methods of Negative Number Representation
    - Signed Magnitude
    - One's Complement
    - Two's Complement
    - Subtraction using Complements (1's, 2's, 7's, 8's, 9's, 10's, 15's, 16's complement)
- **Various Binary Coding Schemes**
  - BCD
  - EBCDIC
  - ASCII
  - ISCII
  - Gray Code
  - Excess-3 Code
- **Concept of Fixed and Floating Point Numbers**
  - Difference between fixed and floating point numbers
  - Concept of normalised numbers
  - Floating point arithmetic (addition, subtraction, multiplication, division)

❑ **Bit map representation of images**

(iii) ***Boolean Algebra***

- Definition and postulates.
- Boolean operations – OR, AND, NOT
- Proof using identities and truth tables
- De' Morgan's Theorems and Basic Principle of Duality
- Deriving truth table from Boolean expression and vice versa
  - Sum of Product (SOP) Expressions (using min-terms)
  - Product of Sum (POS) Expressions (using max-term)
- Canonical form of Boolean expressions and their complements
- Simplifications (Algebraic method, K-map method up to 4 variables)
- Use of Don't Care terms
- Logic Gates – OR, AND, NOT, XOR, X-NOR Gates
- Universal Gates – NAND and NOR Gate
- Basic gates using Universal Gates
- Two Level Circuits
- Combinational Circuits :
  - Half Adder & Full Adder (definition and representation)
  - Full Adder using Half Adders only
  - Half Subtractor & Full Subtractor (definition and representation)
  - 4 bit Adder and Subtractor Circuit
  - Multiplexer (4×1) and De-multiplexer (1×4)
  - Decoder (Maximum 3 bits), and Encoder (Decimal to Binary, Octal to Binary)

B. SOFTWARE AND LANGUAGES : (20 MARKS)

- Definition of Software
- Programming Languages : Concepts of High Level, Low Level and Assembly language
- Types of Software

❑ **System Software**

- Translator – compiler, interpreter, assembler
- Operating systems :
  - Definition and Function
  - Types of OS – Single User, Multi-user, Multiprogramming, Multiprocessing, Time Sharing
  - Booting (cold and warm), Spooling Buffering, Concept of Virtual Memory
  - Directory and file Structure, Path and Pathname
  - Concept of GUI, CUI with examples
  - Using MS DOS (commands and their use – DIR, MD, RD, CD, COPY, CON, MOVE, REN, DEL, TYPE, MORE, ATTRIB, EDIT, DATE, TIME, CLS). Concept of Batch File
  - Using MS Windows OS
  - UNIX OS (Commands and their use – chmod, cd, pr, cp, cat, rn, rndir, ls, vi, mkdir, more, mv, mail, who), Use of Wild Card, File Permission, Concept of Piping, UNIX shell
  - Application Software (definition and example)
  - Utility Software (definition and example)

C. PROGRAMMING : (20 MARKS)

- Concept of Algorithm and Flowchart
- Introduction to C
- Character Sets, Keywords, Constants, Variables, Operators in C
- Data types in C
- Header files
- Input / Output operations
- Control structures
- Loop structures
- Functions (user-defined and common library functions) including recursive function
- Array (one and two dimension numeric array)
- Basic concept of Pointer and String
- Structures
- Problem solving

D. PRACTICAL : (30 MARKS)

❑ **MS-Windows / UNIX / LINUX / Operating System Commands – (5 Marks)**

❑ **Programming in C (Algorithm / Flow Chart, Coding, Execution) – (15 Marks)**

- One program using branching and loop (5 marks)
- One program using Function, Array, String, Structure (10 marks)

❑ **Laboratory Copy (must have minimum 20 programs from topics in class 11) – (5 Marks)**

- 6 programs on control structures
- 4 programs on array manipulations
- 4 programs on string manipulation
- 2 programs on structure manipulation
- 4 programs on functions

❑ **Viva Voce – (5 Marks)**

Contents

1. A Brief History of Computers

- Brief History of Development 1-1
- The Babbage Machine 1-1
- Stored Program Concept 1-2
- Von Neumann Concept 1-2
- Difference between Calculator and Computer 1-3
- Computer Generations 1-3
- SSI, MSI, LSI, VLSI, ULSI 1-6
- Classifications: supercomputers, mainframe, mini, micro 1-6
- Analogue, Digital and Hybrid Computers 1-9
- Computers in Modern Society 1-11

2. Data Processing and Computer Organisation

- Data and Information 2-1
- Computer System 2-3
- Input Devices 2-5
- Output Devices 2-10
- Memory Unit 2-14
- The Processing Unit 2-17
- Storage Devices 2-19
- Communication Bus 2-25

3. Data Representation

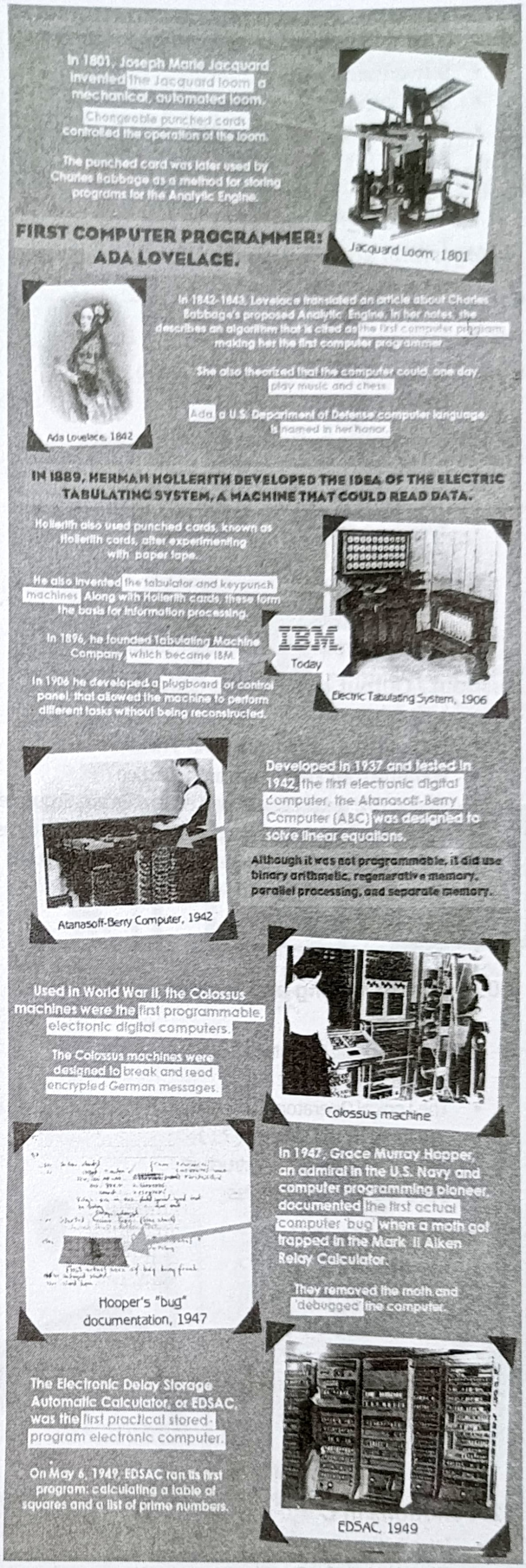
- What is a Number System 3-1
- Conversion of Integer Values from one system to another 3-2
- Conversion of Fractional Values from one system to another 3-10
- Arithmetic Operations in various Number Systems 3-16
- Representation of Signed Numbers using Complements 3-24
- Various Binary Coding Schemes 3-32
- Bit map representation of images 3-34
- Concept of Fixed and Floating Point Numbers 3-35

4. Boolean Algebra and Logic Gates

- Basic Boolean Operations 4-1
- OR, AND, NOT Operations and Truth Tables 4-2
- Switching Circuit Equivalents 4-3
- Boolean Algebra Rules and Proof by Perfect Induction 4-3
- De' Morgan's Theorems & Basic Duality of Boolean Rules 4-8
- SOP, POS, Min Term & Max Term Expressions 4-11
- Canonical form of Boolean Expressions and their Complements 4-15
- Techniques for Simplification: Using Karnaugh Map 4-19
- Logic Gates: AND, OR, NOT, XOR, NAND, NOR, XNOR 4-26
- NAND and NOR Gates as Universal Logic Gates 4-27
- Worked out problems on Logic Gates 4-31

5. Combinational Circuits

- General Description of Combinational Circuits 5-1
- Adder Circuits 5-1
- Subtractor Circuits 5-3
- Multiple-bit Adder and Subtractor Circuits 5-5
- Multiplexer Circuit 5-7
- De-multiplexer Circuit 5-8
- Decoder Circuit 5-9
- Encoder Circuits 5-10



6. Operating System

- Software and its Types 6-1
- Operating System 6-4
- The Disk Operating System (DOS) 6-11
- Windows Operating System 6-22
- UNIX Operating System 6-34
- The Linux Operating System 6-46

7. Algorithm and Flowchart

- General Concepts 7-1
- Different Phases of Programming 7-1
- Algorithm 7-3
- Flowchart 7-5
- Pseudo Code 7-7

8. Programming in C: General Concepts

- Introduction 8-1
- Structure and Components of a C Program 8-1
- Types of Data 8-6
- Constants and Variables 8-8
- Declaring Constants 8-9
- Declaring Variables 8-11
- Writing, Compiling, and Running a program in C 8-12

9. Input / Output and Simple Calculations in C

- Introduction 9-1
- Formatted text output: the printf) function 9-1
- Formatted text input: the scanf() function 9-3
- More on the use of Format Specifiers and Escape Sequences 9-6
- Unformatted text output: the putchar() and puts() functions 9-11
- Unformatted text input: the getchar() and gets() functions 9-12
- Some Basic Operators and Simple Calculations 9-13
- Operators and Data Types 9-15
- Some worked out examples 9-19

10. Decision Making and Branching in C

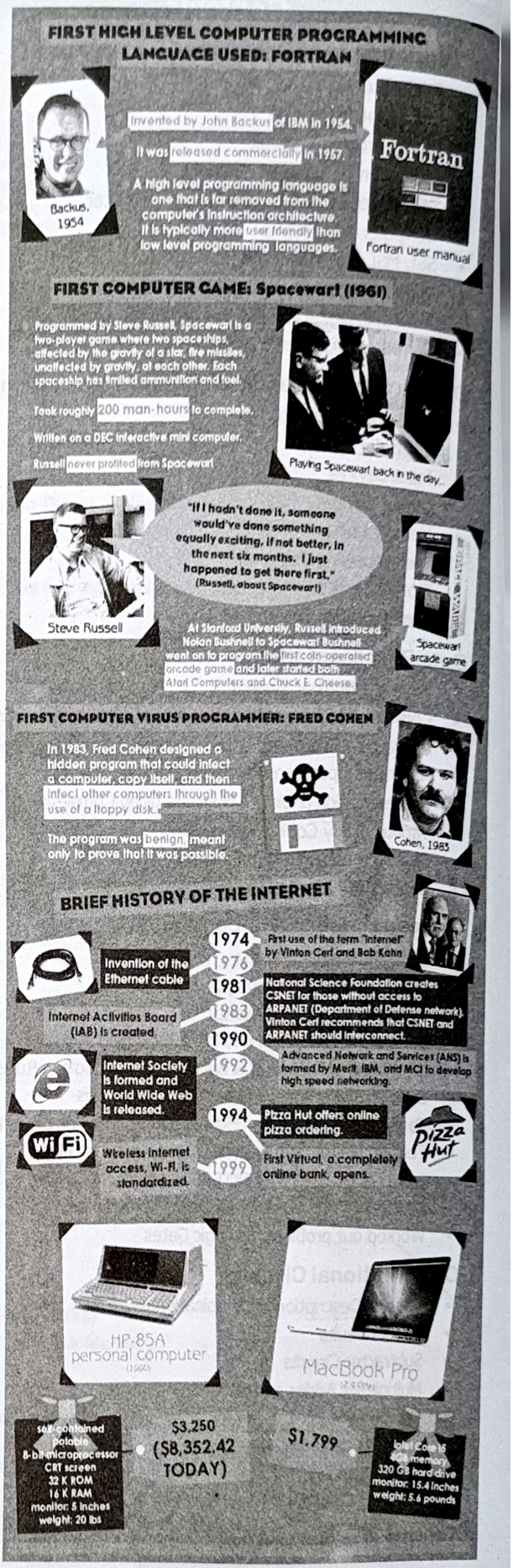
- Introduction 10-1
- The Relational Operators 10-1
- The if-else statement 10-2
- The Logical Operators (AND, OR, NOT) 10-10
- The Conditional Operator (? :) 10-16
- The switch-case-default structure 10-19
- Some worked out examples 10-22

11. Using Loops in C

- Introduction 11-1
- Concept of a Loop 11-2
- The while Loop 11-6
- The break statement 11-14
- The continue statement 11-15
- The do-while Loop 11-16
- The for Loop 11-18
- Some worked out examples 11-24

12. Functions in C

- Introduction 12-1
- The Working of a Function 12-2
- Defining a Function 12-3



- Function Recursion 12-10
- Storage Classes 12-12
- Some worked out examples 12-22

13. Arrays in C

- Introduction to Arrays 13-1
- How to declare an Array 13-1
- How to enter and read data from an Array 13-3
- Multi Dimension Arrays and Matrices 13-5
- Arrays and Pointers 13-9
- Some worked out examples 13-15

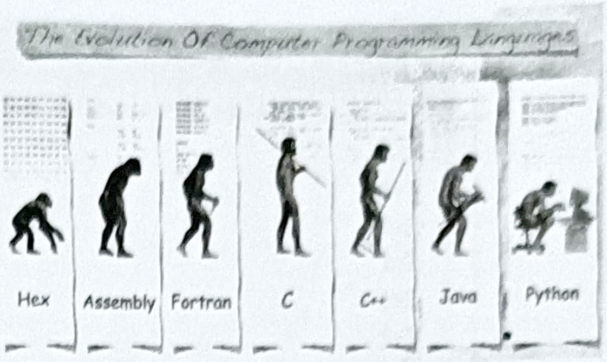
14. Strings in C

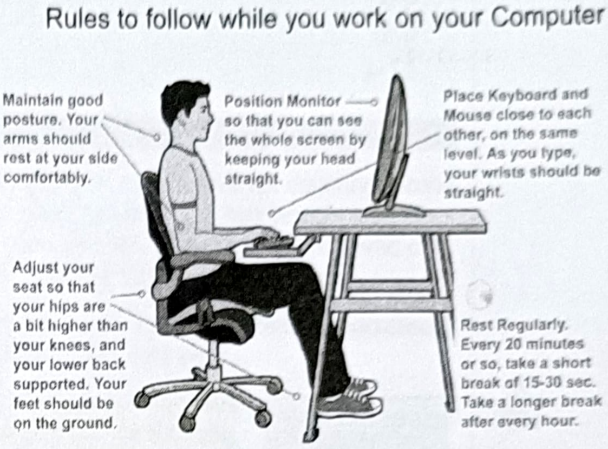
- What are Strings? 14-1
- Entering and displaying Strings 14-2
- Matrix and Strings 14-4
- Some Library Functions available to handle Strings 14-5
- Pointers and Strings 14-10
- Some worked out examples 14-12

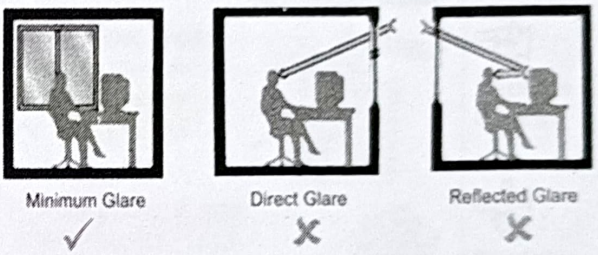
15. Structures and Unions in C

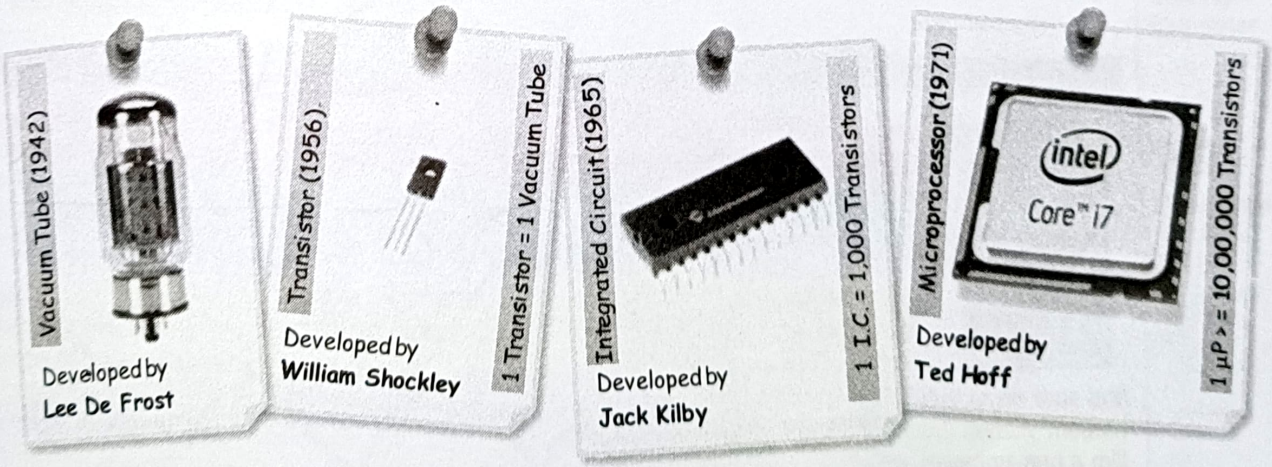
- What is a structure? 15-1
- How to declare a structure 15-2
- How to Enter and Read Data from a Structure 15-4
- Array of Structures 15-5
- Passing Structures to Functions 15-8
- Union 15-9
- Some worked out examples 15-11

- **Appendix-I: List of Programs in the book A1**
- **Appendix-II: Acronyms A4**
- **Appendix-III: ASCII Table A5**
- **Appendix-IV: DOS Commands A6**
- **Appendix-V: UNIX Commands A7**
- **Appendix-VI: Index for C Commands and Operations A7**
- **Answers to MCQs** P1-M-1
- **Last 3 Years' Theory and Practical HS Question Papers with SOLUTIONS** P1-Q-1









Relative sizes and packing density of various electronic components used in computers

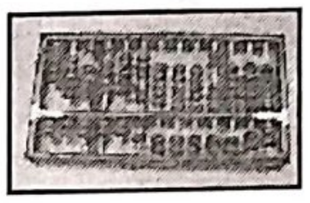
CHAPTER 1
A Brief History of Computers

- Brief History of Development 1-1
- The Babbage Machine 1-1
- Stored Program Concept 1-2
- Von Neumann Concept 1-2
- Difference between Calculator and Computer 1-3
- Computer Generations 1-3
- SSI, MSI, LSI, VLSI, ULSI 1-6
- Classifications: supercomputers, mainframe, mini, micro 1-6
- Analogue, Digital and Hybrid Computers 1-9
- Computers in Modern Society 1-11

1.1 Brief History of Development

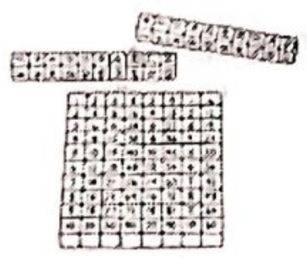
The word computer has being derived from the word 'Computing' which mainly involves counting. Stone Age people used pebbles or made scratch marks on the cave walls to count the number of cattle they had. Whenever a newborn was there, they added another pebble and when someone died they removed one. In this way they kept record of their daily store. Consequently they used their fingers for counting.

All these techniques were slow and difficult to maintain when the total number of entities increased. Subsequently newer techniques and gadgets were developed to assist in their daily work.



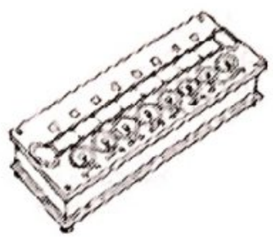
Abacus

The **Abacus** probably is the **oldest counting machine** and was developed around **500 B.C.** It consists of a wooden frame with rods and columns of bead on them for counting. A crossbar usually separates the beads into two sections. By moving the beads one can make additions and subtractions. It is still used in China and Japan.



Napier's Bones

The first device to help in multiplication and division was developed by **John Napier**. Napier designed a system of ivory rods in early 17th century on which were etched several numbers based on the table of logarithms. The rods are better known as **Napier's Bones**. To multiply two numbers one has to simply add two numbers and get the results using the principles of logarithm.



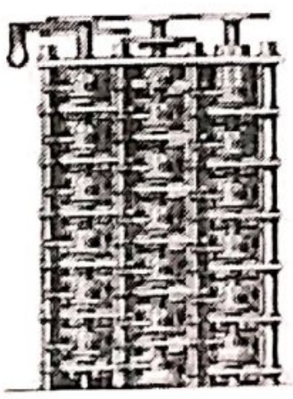
Pascaline

Next came **Pascal's Adding Machine** which was also called the **Pascaline**. It was developed in **1642** by the French mathematician and philosopher **Blaise Pascal** (1623-1662), the 18 year old son of a tax collector. It was a **mechanical calculator** made of gears and wheels and could do additions and subtractions in base 10. This was the first calculator that could automatically do calculations once the numbers were fed to it.

Leibniz Calculator

Later on in 1694, the French mathematician **Leibnitz** (1646-1716) designed an improved **mechanical calculator** by studying Pascal's original notes and drawings. The machine could do multiplication and division as well, along with addition and subtraction.

1.2 The Babbage Machine



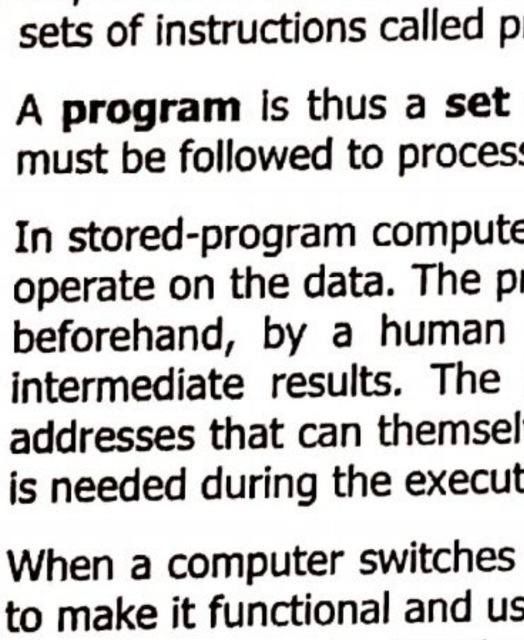
Difference Engine

Charles Babbage (1791-1871), a professor in Cambridge University, is regarded as the **father of modern digital computers**. In 1822 he developed the **Difference Engine** that was later used to produce astronomical tables for an observatory in Albany, New York. The machine mainly calculated **polynomials** (i.e. expressions of the form $a_nx^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2}.....+ a_0x^0$) based on the method of differences.

Later in 1842 he came up with the idea of the **Analytic Engine**, which became his lifetime project. The machine was designed so that the user could feed in a sequence of instructions, and the data on which to work. The machine could then store and execute the sequence of instructions. The hardware technologies required to build it were much ahead of his time and **the final machine could not be constructed successfully**. But his design featured certain modern concepts like storage capacity, a typewriter for an input device, punched cards for storing programs and a mill that could do additions in 1's. **The basic concepts developed by Babbage are the basis for digital computers even today.**

1.3 Stored Program Concept

To create complex designs automatically on woven cloth, a French weaver named **Joseph Jacquard** invented a system of punched cards in early 1880's. The machine operated based on a binary system of communication i.e. holes and no holes. The punched cards controlled the movement of the threads by the presence or absence of holes in the card. Thus an effective means of communication with machines was developed by him and the **Jacquard's loom** worked as per the patterns stored in the cards.



Tabulating Machine

Meantime in the US, the 1880 Census took almost 7 years to tabulate the results. The Census Bureau organised a contest to develop a faster method for the next Census. A mathematician named **Herman Hollerith** was the winner. His **Tabulating Machine** (*picture shown on the left*) was based on the principle of punched cards and his machine could automatically read the Census data punched onto cards. **The data was coded by a combination of punched holes on the card**. A device called a tabulator was used to read the data. When the cards were passed through the tabulator, electrical contacts through the holes were used to read the data. Using Hollerith's machine the 1890 US Census took only 3 years to complete. His **Tabulating Machine Company** was a predecessor to the **International Business Machines Corporation** (IBM).

500 B.C.	Abacus
1617	Napier Bones
1642	Pascaline
1694	Leibniz Calculator
1822	Difference Engine
1890	Tabulating Machine
1944	Mark-1
1946	ENIAC
1949	EDSAC
1950	EDVAC
1951	UNIVAC-1

A *Program* is a set of instructions indicating to the computer the exact sequence of steps that must be followed to process a given set of data

The idea of programming was thus slowly getting developed. Later when computers became more complex, the number of operations to run the computers also increased in number and complexity. The control of input and output devices, transfer of data from the storage devices to the logic unit, processing of data – all these require a series of instructions to be carried out in a particular sequence. All such operations are controlled by sets of instructions called programs.

A **program** is thus a **set of instructions** indicating to the computer the **exact sequence of steps** that must be followed to process a given set of data.

In stored-program computers, memory locations with proper addresses store both data, and instructions that operate on the data. The program consists of step-by-step instructions to process the data, which is prepared beforehand, by a human programmer. The data could be the input for processing or could represent intermediate results. The instructions, which are loaded into the processor for execution, contain data addresses that can themselves be modified by programs if required. No human interaction with the computer is needed during the execution — the controller does all the necessary sequencing and directing of data.

When a computer switches on, it usually reads from a fixed memory location and carries out the instructions to make it functional and useful to a user. This is generally called booting. Hence modern day computers need stored programs to function unlike their classical counterparts where the machines worked on a one to one basis.

1.4 Von Neumann Concept / Architecture

With the beginning of the Second World War much research went into the development of computers that could be used to quickly calculate charts for firing a variety of new weapons used by the U.S. army. Finally the Electronic Numerical Integrator and Computer or **ENIAC** was designed and built at the university of Pennsylvania and went into operation in **1946**. It used the decimal number system, instead of binary. The ENIAC was used for a variety of purposes including scientific research, calculating range tables for missiles, and weather prediction (picture of ENIAC shown on the right).



ENIAC

However before it could perform a task, it sometimes took hours to set up the wiring and the switches, for a program that would execute in seconds. To overcome these problems and to set up a generalised computing machine, the project team members of ENIAC set about developing the Electronic Discrete Variable Automatic Computer or **EDVAC**. The design of **EDVAC** featured more internal memory than any other existing computing device as well as the **conditional control transfer** that allowed the computer to be stopped at any point and then again resume work.

EDVAC

In the mid 1940s **John von Neumann** (1903-1957), a brilliant Hungarian scientist joined the EDVAC team and published a 100-page paper called the "**First Draft**". The draft contained the concepts of the "**five-function computer**" and of the stored program. This concept is also known as the **von Neumann Architecture** of a modern computer.

Von Neumann Architecture

The **five functions**—memory, processing, control, input, and output — detail a design for the **purely mathematical concept** of a **generalised computer** called a **Turing Machine** (concept developed by British mathematician Alan Turing, 1912-1954, also one of the founders of computer science), independent of the type of hardware used. Different kinds of hardware components can carry out these functions—for example, vacuum tubes, transistors, or integrated circuits. The **five components of the Von Neumann Architecture** to implement a Turing Machine are given below:

Turing Machine

1. **An Input Unit**
2. **A central Arithmetic Logic Unit (ALU)**
3. **A central Control Unit (CU)**
4. **A Memory**
5. **An Output Unit**

Based on Neumann's concepts, in **1949** the Electronic Delay Storage Automatic Calculator or **EDSAC** was made and used at the Cambridge University London. Though it was not a binary logic based computer but was the **first stored program electronic computer**.

EDSAC

The term "von Neumann architecture" is also often used to distinguish the sequential approach of computer design from **parallel processing** designs in which there are several processors working simultaneously to solve a problem and from **neural networks** where a pre-written program is not required. The neural network computing system is based on the working of the human brain, which learns to perform functions rather than having functions programmed into it. Unlike the structure of a brain, a conventional computer has a single processing unit for all data. A neural network is more brain-like, with a large number of simple processing units, which parallely process a small proportion of data.

1.5 Difference between Calculators & Computers

The major difference between a general purpose calculator and a computer is that a **calculator can execute mathematical expressions based on inbuilt permanent instructions** and in general is not designed to derive at conclusions based on logical decisions as well. On the other hand **a computer can be programmed** to solve specific problems and deliver the results after mathematical and logical conclusions. (We are not considering programmable calculators, as these also have certain limitations).

A computer can be programmed to take decisions which cannot be done using a normal calculator

For example a calculator can be used to find the average of the marks obtained by students in a school. But to find the total number of students who have secured marks between 80 and 90 in mathematics you need a computer program, as it needs logical decision to be taken by comparing the marks obtained by a student, to the range of marks specified and then arriving at the decision.

Comparison between a Calculator and a Computer:

Calculator & Computer – A Comparison

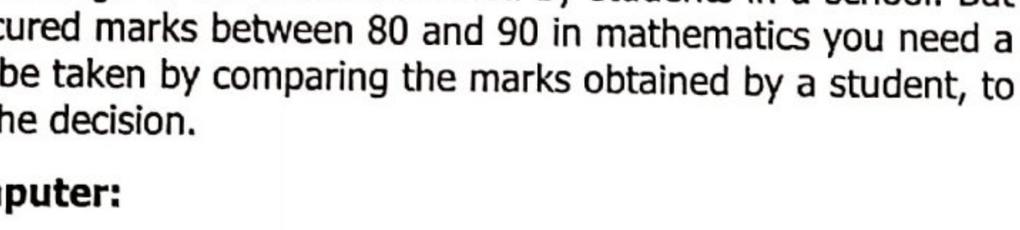
Similarities:
1. Both need **electrical power** to work
2. Both have an **input device** like a keypad or keyboard
3. Both have a **screen** to display the results
4. Both have **memory** to store data and the results
5. Both can do **numerical calculations**

Differences:
1. A computer can do bigger and **more complex calculations** than a calculator
2. A computer has more memory when compared to a calculator
3. A computer can do other tasks in addition to calculations, and can **do several jobs simultaneously**
4. A computer can be **programmed in various languages** whereas a calculator has no such facilities

1.6 Generations of Computers

Computer Generations

Depending basically upon the **kind of electronic switching device used for storing numbers,** we can divide the development of **commercially available** computers into five different categories or **Generations**. Earlier the term generation was used to indicate the **different hardware technologies**, but
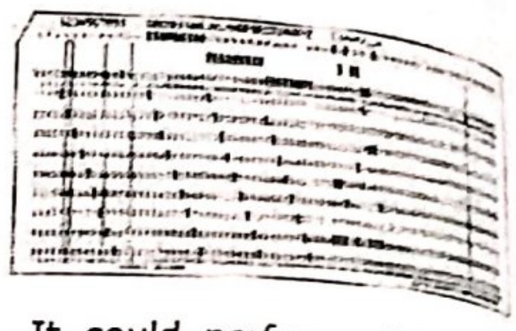


Vacuum Tube Transistor Integrated Circuit (IC) Microprocessor

presently both hardware & software technologies together form the basis of the computer generations. The figure above shows the **relative sizes of the different components** used in each generation.

First generation computers used Vacuum Tubes and consumed high electrical power

• First Generation (1942-1955) – The age of the VACUUM TUBE



The earliest form of computers built towards the end of 1930's like the **Harvard Mark 1** (also called the Automatic Sequence Controlled Computer or ASCC) operated by the **use of relays** and was an **electromechanical device**. It was the first fully automatic information-processing machine and could complete lengthy computations without human intervention. The machine took **3-5 seconds per calculation** and had an **overwhelming size of 51 feet (length) by 8 feet (height)**. It could perform all the four arithmetic operations and could handle logarithms and trigonometric functions. It **used punched cards** for data input and electromechanical switches or **relays** for storing data. These were very slow and could operate at a very slow rate of about a hundred times a second.

The ENIAC contained about 17,450 vacuum tubes and 5 million hand soldered joints. It weighed more than 25 tons and consumed nearly 150 kW of electrical power



The **vacuum tube** (picture on the right) invented in **1906** by the American Physicist **Lee De Frost** formed the basis for the first electronic computers and soon replaced the electromechanical relays. **Vacuum tubes did not have any moving parts** and could switch several thousand times a second and worked much faster than a relay. However it consumed huge electrical power.



While the **ENIAC** commissioned in 1946 was the world's first general purpose electronic computer, the **UNIVAC-1** (**UNIV**ersal **A**utomatic **C**omputer, shown in the picture above) was the **first general-purpose electronic digital computer designed for commercial use**. It was started in 1946 and completed in 1951. It could do about 8000 additions/sec and about 250 divisions/sec. The UNIVAC's input consisted of magnetic tape. There were about 3000 installations of first generation computers.

Second generation computers used Transistors and required much less power

• Second Generation (1955-1964) – The age of the TRANSISTOR



The introduction of **semiconductor components** marked the beginning of the second generation of computers. Vacuum tubes consumed a large amount of power. **Transistors** invented in 1947 at the Bell Laboratories (see picture on right) were made of semiconductor materials and required no heating element or vacuum and effectively **required much less power** to operate. Transistors could **switch also at a much faster rate** of several hundred thousand times a second. The **IBM 1401** and **IBM 1620** became universally accepted machines during this period.

Third generation computers used Integrated Circuit technology, which allowed hundreds of transistors to be packed into a single silicon chip

• Third Generation (1965-1970) – The age of the INTEGRATED CIRCUIT (IC)

With the use of transistors, though the general size of the computing machines decreased but as computers became more and more powerful, more and more transistors needed to be used. Like other electronic components this meant more and more transistors needed to be soldered into the circuit boards and the likelihood of faulty wiring also increased. The number of transistors being large and in the order of 10,000, the heat generated was enough to damage the computer's sensitive internal parts.



This led to the development of the **integrated circuit** (**IC**) by engineers at Texas Instruments **in 1959**. The IC combines the electronic components onto a small wafer of silicon only several millimetres thick. **Integrated Circuits began to replace discrete transistor circuits used in second-generation machines** resulting in substantial reduction in physical size and cost. By 1965 semiconductor (IC) memories began replacing ferrite core memory designs. Important members of the third generation of computers include the **IBM's System-360** series, UNIVAC 1108, RCA 3301, GE 645, Honeywell 200 series etc.

Fourth generation computers used Microprocessors where the entire CPU was fabricated on single silicon chip

• Fourth Generation (1971-1985) – The age of the MICROPROCESSOR (µp)



With the evolution of IC technology more and more transistors were packed inside a single silicon chip. **VLSI & ULSI Technology helped to fabricate ICs with thousands and later millions of transistors on a single IC Chip** and marked the basis for the fourth generation of computers.

Initially the CPU of an IC based computer was of the size of a refrigerator with several ICs

(each with a particular function), several transistors and other electronic components used to make the CPU. Later, instead of using various individual parts, **an entire CPU was fabricated on a single silicon chip** that could also be mass-produced at a very low cost. Thus millions of transistors were packed onto the single IC in the process, to produce a **microprocessor**. The **first microprocessor chip, the 4004,** was developed by Intel **in 1971**, which ushered in the first microcomputer – the predecessor to the modern day PC. The 4004 could process 4 bits of data in parallel. It was also the first commercially available microprocessor that could be programmed for different tasks.

In a microprocessor since the transistors are smaller and closer together, they can also **turn on and off faster**, **at tens of millions of times a second**, resulting in higher performance rates. The Intel 8085, 8086, 80286, 386, 486, Pentium etc. are microprocessors used in PCs of this era.

Fifth generation computers are desianed to have Artificial Intelligence

- **Fifth Generation (1985-Present) – The age of ARTIFICIAL INTELLIGENCE**

Fifth Generation Computers

The computer technology of the fourth generation and the fifth generation **do not have a distinct borderline**. But the major aspects of fifth generation computers are to make computers with **artificial intelligence**. The **computers should be able to learn from their mistakes**. Fifth generation computers will be able to accept spoken word instructions and imitate human reasoning. They are used for speech analysis, pattern recognition and other complex processing tasks.

The new technologies that are being used include the concept of **parallel processing** (with a system of many CPUs to work as one) and **neural network** which is based on the working of the human brain. This is a major diversion from the von Neumann architecture containing a single central processing unit design.

Characteristics of the Different Generations of Computers

Characteristics of different generations

Generation	Period	Characteristics
First	1942 to 1955	• **Vacuum tubes** were used to build these computers (the **ENIAC** for example contained about 17,450 vacuum tubes!) • **Very large in size** (ENIAC required about 28,000 sq. ft. of area) • Very **high power consumption** (ENIAC required about 150 kW of power) • **Generated much heat** • Slow processing **speeds of the order of milliseconds** • These were **programmed in machine language**
Second	1956 to 1964	• **Transistors were used** to build these computers • These were **smaller, cheaper, faster**, more reliable and energy-efficient than their predecessors • Computers were **commercially available** for business and science • **Magnetic core based memories** used instead of vacuum tubes • Machine-independent **High Level programming languages** like COBOL, FORTRAN, etc. were introduced to simplify programming • **Floating Point Operations** were widely used • The **processing speeds** of these computers were in the order of **microseconds**
Third	1965 to 1970	• **Integrated Circuits were used** instead of transistors to build these computers • **Smaller, cheaper, faster**, more reliable and energy-efficient than their predecessors • **Semiconductor memories** used instead of ferrite core memories • Direct Access Storage Devices (DASD) or **disk drives were introduced** • The **use of an Operating System** allowed machines to run many different programs at once with a central program monitoring and coordinating the operation of the computer • **Mass production** of small low-cost computers called minicomputers made them suitable for a wide variety of applications including industrial control • The **processing speeds** of these computers were in the order of **nanoseconds**
Fourth	1971 to 1985	• **Microprocessor technology** used to build these computers made them smallest in size • These computers were the **cheapest & fastest** among all generations • These minicomputers came complete with **user-friendly software packages** (like word processors etc.) for even non-technical users • As smaller computers became more powerful, they could be linked together, or networked, to share resources. The **concept of LAN & WAN developed**.

The computer in a modern day cell phone has more processing power than all the computers in the Apollo-11 Lunar Lander that helped 2 men to land on the moon in 1969

Generation	Period	Characteristics
Fifth	1985 to Present	• Based on the concept of **parallel processing** and **neural networks** • Use **PROLOG** (**PRO**gramming in **LOG**ic language) **for programming** • Computers to possess **basic intelligence** to make decisions • Capable of being a **Knowledge Information Processing System (KIPS)**

1.7 SSI, MSI, LSI, VLSI, ULSI

Integrated Circuits

The terminology SSI, MSI, LSI, VLSI and USLI are used to **indicate the total number of active components or transistors that are being packed into a single IC chip**. With the advent of IC technology initially up to 10 components could be fabricated inside a silicon chip and the technology was termed as **S**mall **S**cale **I**ntegration. Subsequent developments led to the following technologies:

- **SSI** or **S**mall **S**cale **I**ntegration: Contains **less than 12** integrated components like transistors, diodes, resistors per chip.
- **MSI** or **M**edium **S**cale **I**ntegration: Contains about **12 to 100** integrated components per chip.
- **LSI** or **L**arge **S**cale **I**ntegration: Contains **more than 100** integrated components per chip. Usually MOS transistors are used instead of bi-polar transistors for making an LSI design as these are much more compact than bi-polar transistors.
- **VLSI** or **V**ery **L**arge **S**cale **I**ntegration: Contains **up to hundreds of thousands** of components per chip. All modern day chips use this technology.
- **USLI** or **U**ltra **L**arge **S**cale **I**ntegration: Containing **more than 1 million** components on a single chip. Modern day microprocessors like the Pentium, CORE**i** etc. use this technology.

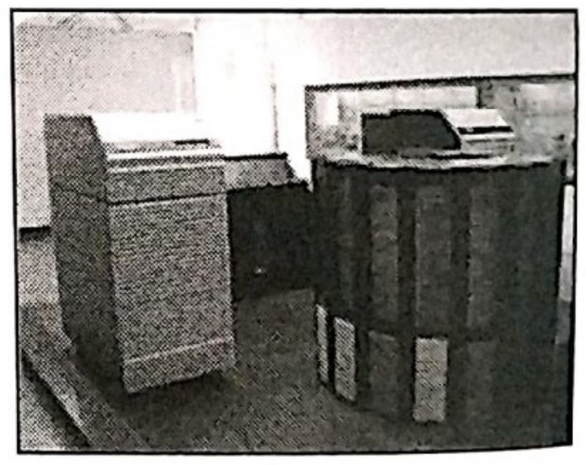
1.8 Classifications: supercomputers, mainframe, mini, micro, special

Classification of Computers

Depending upon the size and capability of the computers these can also be classified as supercomputers, mainframe, minicomputers, and microcomputers/desktops. Thus on one end of the spectrum we have the supercomputers and on the other end we have the personal computers or desktop PCs. At present, with the rocketing speed at which technology is developing and more and more transistors are packed into a single chip, the processing power of the present day computers are increasing by leaps and bounds. With more people buying computers than a decade back, the prices of computers are also dropping. In this scenario, the **distinction between the above different classes is also narrowing down** or has become relative. The power of a present day high-end personal computer is comparable to the processing power of a supercomputer a decade ago. Still the basic distinctions between the above types are given below for academic interest.

Supercomputers are the most powerful and expensive computers available

- Supercomputer:



Supercomputers are large computers used mainly in Scientific and Research organisations for **processing complex scientific data that require huge processing power**. These are used in research work like in atomic collidors, weather forecasting, astronomy, speech analysis, code breaking etc. Supercomputers **use parallel processing** to solve complex problems faster than an ordinary computer. These are also the **most powerful and expensive** computers available. The price of a supercomputer can be of the order of Rs. 100 crores.

A present day iPad2 has similar computing power as Cray-2 super-computer in 1985

It would take 60,000 years for 1,000 people working at a rate of 1 calculation per second to complete the calculations that the Titan super-computer built in 2012 can do in one second

The Hopper supercomputer contains about $2x10^5$ GB of RAM

The **speed of a supercomputer** is generally measured in units of number of **fl**oating-point **o**perations **p**er **s**econd or **flops** that it can perform. The **Cray1 Supercomputer** was the first computer capable of **performing at over 10^8 flops**. Although faster machines have now been built, Cray1 continues to be used and serves as the informal unit of measure for newer supercomputers, some of which now equals 10^6 'Crays'.

The Chinese supercomputer **Tianhe-2** is at present (as on 2014) the world's **fastest supercomputer**. It can do about $34x10^{15}$ calculations per second (speed 34 PetaFlops). Param 8000 built in 1990 is considered India's first supercomputer. The **Param Yuva II** supercomputer built in 2013 can do $524x10^{12}$ calculations per second (speed 524 TeraFlops). In comparison, a Core i7 desktop PC can do $7x10^9$ calculations per second.

The **Human Genome Project** meant for an understanding of the entire genetic blueprint of a human being was an example of the use of supercomputers in recent years.

- **Mainframe-computers:**

Supercomputers have limited applications and are too expensive for normal use. Mainframe computers on the other hand are the largest type of computers after supercomputers, which are put to common use (picture of 1990 Honeywell-Bull DPS-7 mainframe shown on the right).



Mainframe computers are used in large organisations such as banks, railways, airways, hospitals, etc. that need frequent access to the same information and quickly process large number of transactions on-line. The information are usually centrally stored in one or more huge databases. This requires **computer systems with massive data storage and processing capabilities.** Mainframe systems are special type of computer systems that meet these criteria.

Mainframes are used in large organisations that frequently access same information and quickly process many transactions on-line

These are expensive, large, high-speed multi-user systems and are usually housed centrally in several large cabinets. Mainframes often **support thousands of simultaneous users** who gain access through specialised terminals or from smaller computers. Nearly all mainframes have the **ability to run multiple operating systems**. Price of a mainframe can range from Rs. 20 lakh to several crores.

Mainframe computers hold approximately 70% of the entire data stored in this planet

So far no computer virus has attacked any Mainframe computer

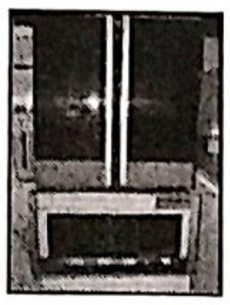
- **Minicomputers:**

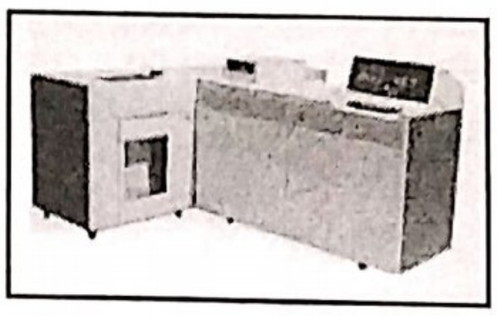
These belong to a class of multi-user computers which make up the middle range of the computing spectrum, **in between the mainframe computers and the personal computers**. In modern times such machines are sometimes referred to as workstations (as with Sun Microsystems) and servers.

These were a **product of the third generation of computers** and were developed during the 60's. They usually took up one or more cabinets the size of a large refrigerator or two, compared with mainframes that would usually fill a room. These computers had **large memory capacity,** faster input/output devices, ran in full **multi-user mode**, used operating systems like VMS and UNIX with **multitasking capabilities.** These often used versions of BASIC for application program development. They were used when the volume of data to be processed was large. Typical applications included **financial accounting**, payroll calculations etc. in an office.

Mini-computers were in between mainframes and personal computers and were used to process large volumes of data in an office.

The **first successful minicomputer** was DEC's 12-bit **PDP-8** (first picture shown on the right) launched in 1964 at a cost of about Rs. 8 lakhs. **IBM's System38** (second picture shown on the right) introduced in 1978 is an example of another important minicomputer series. The legacy of minicomputers can still be found in the CPU and OS design of today's PCs, which have evolved by integrating the basic features of a minicomputer.





- **Microcomputers:**

The microcomputer came after the minicomputers. These computers were **small enough to be placed on a desk and cheap enough to be owned by an individual**. Home computers (the Apple and the IBM PC), workstations, laptops etc. are examples of microcomputers.

Micro Computers

The advent of the microprocessor in 1971 paved the way for microcomputers. The many distinct components of a minicomputer's CPU were fabricated on a single integrated chip to form a microprocessor. The earliest microprocessors were basically general-purpose calculator chips. However, with the rapid advancement of microprocessor design, microcomputers grew smaller, faster and cheaper. This made the computer affordable to the general people and resulted in its huge popularity.

Most of the equipment used by a microcomputer is placed within a single case, with some parts like the keyboard, monitor, mouse etc. being connected externally. They were **designed as a single user system**. However, some can work as a server computer running a multi-user operating system like UNIX, and may cater to several users concurrently.

Micro-computers are small enough to be placed on a desk and cheap enough to be owned by an individual. Examples include desktops, laptops.

The first generation of microcomputers were generally shipped as kits and needed to be assembled before these could be used. This required technical expertise on the part of the user. However the second generation of microcomputers, also known as **home computers**, became highly popular as these were basically designed **for use**



by non-technical people. Home computers are also known as **desktop computers** as these can be accommodated on a desk, unlike mainframes or minicomputers, which require large room space. Home computers **generally used a 8 bit microprocessor** and a large variety of 8-bit home computers were marketed during the 80s. These computers often integrated the CPU with the keyboard (like the Commodore 64, picture shown on right) and used the TV as the VDU and audio cassettes as secondary storage media.





The **Altair 8800** was the **first successful microcomputer** built around 1975. The **Commodore-64** was a very popular microcomputer and is regarded as the best selling home computer of all times. Other home computers included the **Apple II,** and the **IBM PC** introduced in 1981 (pictures of the Apple II and the original IBM PC are shown on right). The IBM PC used the **Intel 8088** microprocessor and came with the MS DOS operating system.

A short discussion on the **different microcomputer based computers** is done below:



Personal Computers

- **Personal Computers:** Personal computers are **basically home computers.** The introduction of the **IBM PC** in August 1981 revolutionised the history of computing. Due to the popularity of the IBM PC, within a few years other manufacturers were copying the IBM design and creating replicas of the IBM PC. These computers were in turn also termed PCs, instead of home computers. For this reason **the term PC is nowadays used to indicate the range of home computers that include IBM and IBM compatibles.** However, strictly speaking the word PC is a brand model name by IBM and hence should not be used to refer to all personal computers. For example the Apple's Macintosh line of computers is a personal/home computer but does not follow the IBM PC design.

  A **present day personal computer** is comparable or even better than a mainframe or a minicomputer of yesteryears. Current specifications (as of 2014) of a personal computer include multi-core **processors** (Core i5, I7 etc.) with above 3GHz clock speeds, 4 GB **RAM**, 500 GB **Hard disk drives**, DVD **combo drive, Blu ray** drive, various **ports** like parallel, serial, and USB, inbuilt sound, graphics, and network cards, etc. The CPU in a desktop computer nowadays can even be integrated with the keyboard-display.

  Personal computers are normally operated by a single user at a time. Various types of works that are performed by a user on a personal computer include word processing, internet browsing, e-mail and other digital messaging, multimedia playback, video game play, computer programming, digital image and video editing, etc. Manufacturers also make special OS and application software to be used with a PC.

A **workstation** is a high end desktop microcomputer especially with respect to graphics and processing power.



Workstations

- **Workstations:** Similar to a PC, a workstation is a **high end desktop microcomputer.** Workstations (*picture shown on the right*), usually offer higher performance than a personal computer, especially with respect to graphics, processing power and multitasking abilities. However today's personal computers are more powerful than the top-of-the-line workstations of one generation before. As a result workstations are becoming increasingly more specialised and are usually nowadays **used for manipulating complex data related to 3D graphics animation**, engineering simulation, complex mathematical plots, weather forecasting etc. For example the 3D animations used in blockbuster movies like Jurassic Park and Titanic was done in workstations.



  Workstation hardware can consist of a high end **64-bit processor** like an Intel Itanium-2, large amounts of memory, and special 3D accelerator cards. A workstation often supports 17" or 21" high resolution **dual displays** to double the viewable information. Other normal accessories include input devices like keyboard, mouse, scanner etc. and output printers. Workstations usually run the UNIX operating system or some of its variants. The **Sun SPARCstation 1+** is an example of a workstation of the 90s.

The first successful laptop was the Japanese Kyocera Kyotronic 85, introduced in 1983

Numerous ThinkPad laptops have been used in Space Missions like the Space Shuttle Endeavour's flight in 1993 to repair the Hubble Space Telescope

- **Notebook PC & Netbooks:** These are portable battery operated computers and are used by people who need mobile computers in their field of work, like business executives, sales people and engineers. These are **ideal for people on the move** or for giving corporate presentations. The size of a notebook computer is such that it can easily **fit inside a briefcase and are light in weight** to carry. These are also called **laptop** computers as these can be placed on the lap while working!



  Notebook computers have a full size keyboard, an LCD/TFT display, trackball or touch-pad mouse, floppy

disk drive, CD-ROM/DVD drive, Serial and Parallel ports, USB ports etc. A typical specification can include a high-end processor like an Intel Core-i5, 500GB of Hard disk, 4 GB RAM, DVD combo drive, 15" TFT screen etc. all packed into a small briefcase like unit.

Netbooks are also portable computers but **smaller than traditional laptops**. The cost of a netbook is lesser than a laptop but the internal components are less powerful than regular laptops. These typically have screen sizes of 10 inches. Netbooks are mainly used by college students for doing assignments.

- **Palmtop Computers**: These are **small battery operated computers**, that can **fit inside the palm of a person** and are also called **Personal Digital Assistants** (or PDAs). These computers are mainly used by business persons to store contact information of individuals, perform task scheduling or to browse through emails using an Internet connection.

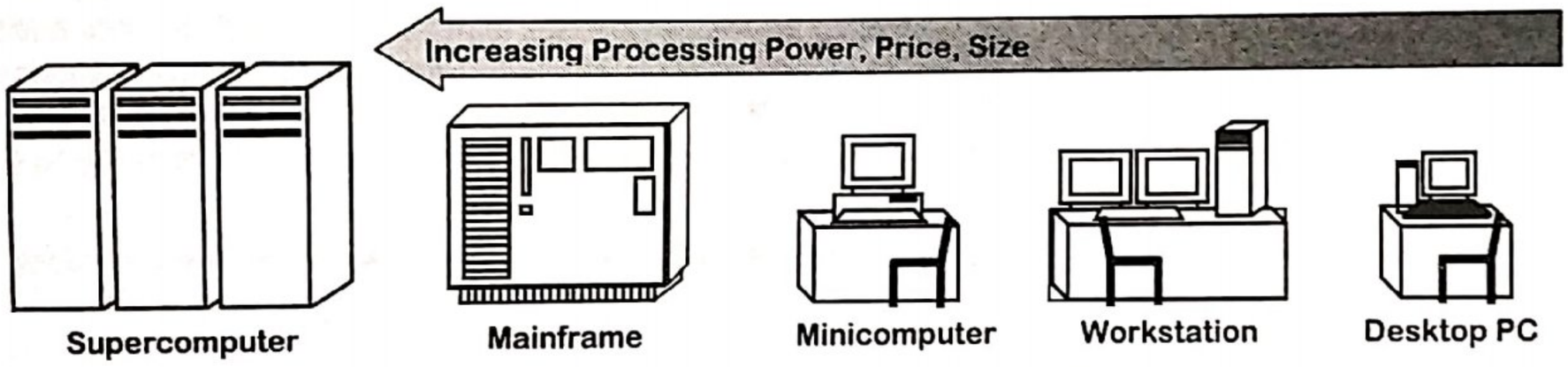


Palmtop Computers

Differences Between A Microcomputer And A Supercomputer

Microcomputer / Personal Computer	Supercomputer
These are small computers **commonly found in home and office** and are used for running application programs for doing accounting, word-processing, database management, and similar jobs.	These are large computers used **mainly in Scientific and Research organisations** for processing complex scientific data, weather forecasting, and other processes that require huge processing power.
These usually use a **single processor** along with multitasking capabilities.	These use **parallel processing** with multiple processors to solve complex problems faster.
The **cost** of micro computers is **low** and can cost from several thousand rupees to up to a lakh.	Supercomputers are the **most powerful and expensive** computers available and costs several crores of rupees.
Example: Personal Computer with a CORE i3 processor.	Example: Cray, Deep Blue, PARAM 10000

Difference b/w Micro & Super Computer



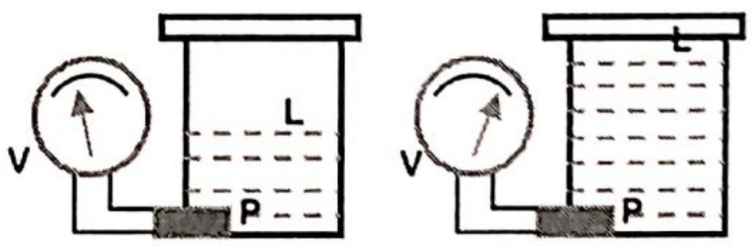
1.9 Analogue, Digital and Hybrid Computers

Based on the **working principle**, a computer may be classified as an analogue or a digital one.

- Analogue Computers:

An **analogue computer** is a form of computer that **uses a particular physical property**, like the electrical or mechanical characteristics of a device to solve a problem. In an analogue device **continuous variation of a process** (like the variable level of water in a tank) is usually **measured using a continuous variation in a physical quantity** (like voltage, current, pressure, torque etc). Subsequently calculations are done based on the measured value of the physical quantity and the result or the output is generated in the form of another voltage or pneumatic signal.

Analogue Computers



As an example of an analogue device, suppose the level of water in a tank is to be indicated. A pressure sensitive device like a piezo-electric crystal or a strain gauge can be used to convert the water pressure, which is proportional to the level of the water in the tank, into a corresponding voltage signal. A voltmeter can then be used to display the change in voltage as a result of the change in the water level. As the **water level changes continuously, the voltage generated also changes continuously** leading to a continuous change in the voltmeter reading. Such an arrangement is shown above with **P** as the pressure sensing device and **V** as the voltmeter. Notice, as the water level **L** has changed, so has the voltmeter reading. As the **voltmeter reading is proportional to the water level**, the voltmeter can be calibrated in terms of height, to indicate the water level.

An Analogue Computer uses a given physical property, like the electrical or mechanical characteristics of a device to solve a problem.



Examples of analogue computing devices include Voltmeters, Energy Meters, Speedometers of Automobiles, Analogue Square Root Extractors, **Slide Rules** etc. These were widely used in Chemical Process Plants, Power Plants, Flight Simulators etc. till digital devices, which are more accurate, slowly took over. The picture of the Polish Analogue computer **ELWAT** is shown to the right.



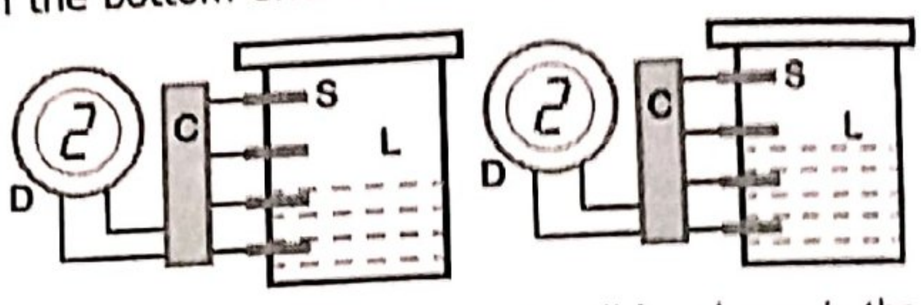
Digital Computers

- Digital Computers:

A digital device **uses discrete electrical voltage levels** to encode a real life situation or process. As such **continuous processes** or variables (*like continuous variation of water level in a tank*) are **encoded as discrete states**, which are then expressed as a set of numbers. In general binary numbers are used to express these states and the circuitry of a digital device consists of millions of tiny electrical switches, which are either in an ON (1) or OFF (0) state representing these binary numbers and **all calculations are carried out using these binary numbers**. For example if the colour of an object is purple, then the colour can be expressed as a combination of three 8-bit binary numbers representing the three primary colours, as Red=10101111 (175), Green=00110110 (54), and Blue=11100000 (224). Thus a **physical property like colour with innumerable shades is quantised** and expressed as a combination of finite numbers. This is how a modern day digital picture scanner or a digital camera works.

Digital Computers *use discrete electrical voltage levels to encode a real life situation or solve a problem.*

As an example of a digital device, consider measuring the water level in the tank. Now instead of using an analogue pressure sensor, **four different sensors S** are used (*shown in the figure on the right*). As the water level **L** rises, each sensor gets activated, starting from the bottom one. In this way we can measure 4 distinct water levels corresponding to the positions of the 4 sensors. These can then be expressed as a set of 4 binary numbers as (00, 01, 10, 11). The digital circuit **C** then computes the level and displays it in the digital display **D**. However, for two different water levels if the number of sensors activated is same, then the same output will be shown. The diagrams above represent this fact, as the same reading i.e. 2 will be shown in the output meter, as only two sensors are activated in both the cases. Thus the **accuracy of a digital device is limited by the number of different discrete states that it can measure**. The accuracy of such a device may be increased by increasing the number of states that it uses to represent a particular measurement (in this case by increasing the number of sensors).



Examples of digital devices include **modern computers**, digital cameras, scanners, thermometers, clocks.



Hybrid Computers

- Hybrid Computers:

There is a third class of devices which **combine the features of both analogue and digital computers**. These are known as hybrid computers. Hybrid computers usually use an analogue computer front-end, the signal output from which is then fed into a digital computer for further computation. A **hybrid computer is usually substantially faster than a digital computer**, but can supply a far more precise computation than an analogue computer. It is useful for real-time applications requiring such a combination like weather system computation etc.

Hybrid Computers *combine the features of both analogue and digital computers to solve a problem*

Examples include the body parameter-monitoring unit in an ICCU. The analogue section of such a computer measures a patient's vital signs like temperature and blood pressure, converts these to numbers and passes these numbers onto the digital sections of the computer for proper analysis.



Difference between Analogue & Digital devices

Differences Between an Analogue and a Digital Computer

Analogue Computer	Digital Computer
Computations are performed using **continuous variations of physical properties** like electrical resistance, voltage, frequency, pressure etc.	A digital device **uses discrete electrical voltage levels** to encode a real life situation or process.
Calculations are normally done in real time i.e. simultaneously, making the **process faster**.	Digital computers have **relatively large delays** as calculations are done in a sequential manner.
The **accuracy of analogue computers is not very high**. Analogue computers calculate by measuring, and the net accuracy depends upon the accuracy of the individual internal components of the devices and on the visual limitation in observing the output.	Since different fixed voltage levels are used to represent binary numbers, **the accuracy of digital devices is very high** and is rarely being affected by the accuracy of the individual components of the device.

Analogue Computer	Digital Computer
Analogue computers can either be mechanical or electrical in nature. Accordingly these are **made up of mechanical parts or electrical circuit components** like resistances, capacitors, inductors etc.	The circuitry **consists of millions of tiny electrical switches**, which are either in an ON (1) or OFF (0) state representing binary numbers and all calculations are carried out using binary numbers.
Computations are performed as **variations** of different physical quantities.	**Computations** are performed using **binary numbers** and binary arithmetic.
Examples of analogue computing devices include Energy meters, Speedometers of cars, Square Root Extractors, **Slide Rules** etc.	**Examples** of digital devices include all **modern day computers**, digital cameras, scanners, thermometers, clocks etc.

1.10 Computers in Modern Society



Computers in Modern Society

In this age of information, computers are the forerunners in providing information to people. They also help to manage various kinds of information such as accounts, employee data, sales data, etc. and help in research and development. In addition, the advent of the Internet has made the computer a common communication tool with its e-mail and chat facilities. These **various uses** of computers are discussed below:

1. **Airports and Railway Stations:** In an airport computers help in providing information to passengers about flights, monitor the flight paths of different airplanes, book tickets etc. In a railway station computers help in providing information to passengers about trains, keep track of the trains in motion, book tickets, enable fast and efficient reservations. In both places computers are also used to maintain an up–to-date data of staff.

2. **Banks:** Computers in banks are used to keep track of customer account information like deposits, withdrawals, balances and transfer of money from and to other banks, made by the users. They are also used in Online Banking, Tele-Banking and Automated Teller Machines.

3. **Factories and Offices:** In a factory computers are used to keep an account of the inventory i.e. the list of raw materials in stock, the details of finished products and to maintain accounts. Computers connected to special purpose machines called CNC machines are used in factories to control the machinery they use to produce their products, help in quality control, etc. In an office computers are used to help in every aspect of work – writing letters, calculating salaries, keeping records of the employees, customers etc. sending emails, and maintaining a network of computers for communication between different computers.

4. **Homes:** At the home end computers are used for entertainment purposes like listening to music, viewing video, editing digital photographs, surfing the Internet and sending e-mails etc. Libraries of interactive CDs on various subjects are also used as an educational aid for students. Teaching aids like Computer Based Training (CBT) CDs are also used by students. Playing computer games at home is a favourite pass time for the young generation.

5. **Hospitals:** Special purpose computers are used to scan and analyse patients and diagnose their diseases. Moreover computers come handy in keeping patient records for ready reference.

6. **Schools and Colleges:** In schools it is used to keep track of student records like their personal data, attendance information, examination results, etc. and teach various subjects using various computer aided teaching tools like multimedia CDs. The use of the Internet adds as a source of information to students and acts as an e-library.

7. **Multimedia:** Multimedia describes a wide range of applications that involve the **use of various media formats like video, sound, picture, text etc.** All these media formats can be combined to form a presentation by the user.



Multimedia

A present day **multimedia system can consist of** a high end computer system with speakers, interface cards to input audio and video data from audio and video input devices, musical instrument interface cards and accessories like printers, scanners, digital-cameras, web-cams etc.

Multimedia data has a specific rate at which it needs to be delivered to the user. For example audio and video data needs to be delivered at real time to a user. To meet this timing requirement, the **multimedia data is often compressed** before it is delivered to the user for playback.

Apart from one's local file system, multimedia files may also **reside in remote servers** as in the case of the Internet. On request, these files are delivered to a client across the network using a technique known as **streaming.** Streaming multimedia can be viewed by the user in real time.

The variety of areas where multimedia is used today includes:

a. **Application Programs:** Several application programs use multimedia features. Apart from text, **e-mails** can now also **include pictures, audio, and video clips**. On-line chatting programs also have options to integrate multimedia features.

b. **Desk Top Publishing:** At present a person with a PC on his desktop and proper application software and a desktop printer can design and **publish multicoloured literary works**, brochures, posters etc. including text and pictures, without even going to the press.

c. **Education:** Computer based teaching tools like **interactive training CDs**, e-books, e-encyclopaedias, e-dictionaries etc. make heavy use of multimedia features to make learning both interactive and fun.

d. **Film industry:** Cartoon **films with 3D visual effects**, sound, and animations, special effects for stunt scenes, making of imaginary creatures, all use various multimedia tools.

e. **Computer Games:** Multimedia features are used to **create interesting interactive games** using software like Flash, Director etc.

The Fact File

- The Chinese had the Abacus as early as 1300 BC
- **In 1617 John Napier of Scotland invented a calculating device called the Napier's Rods.**
- In 1642 Blaise Pascal of France invented a calculating machine called the Pascaline.
- **In 1694, a German mathematician and philosopher, Leibniz created a machine that could also multiply.**
- Next, a Frenchman named Colmar invented the Arithometer, a machine that could perform the four basic arithmetic functions of addition, subtraction, multiplication and division
- **In 1801 the Punch Card was invented by a Frenchman called Jacquard.**
- In 1823 a calculating machine called the Difference Engine was developed by an English mathematician, Charles Babbage. Later in 1834 Babbage developed an even better machine called the Analytical Engine.
- **In 1887, an American inventor, Herman Hollerith, made the punch card reader. He founded the Tabulating Machine Company in 1896 which later in 1924 became International Business Machines (IBM).**
- In 1941 Howard Aikeen of the Harvard University at USA and engineers at IBM together developed the first computer called the Harvard Mark–1. It was a large device and worked electro-mechanically.
- **By 1948, the invention of the transistor by Shockley greatly changed the computer's development.**
- In 1958 Jack Kilby, an engineer with Texas Instruments developed the integrated circuit (IC). As a result, computers became ever smaller.
- **USLI or Ultra Large Scale Integration contains more than 1 million components on a single chip. Modern day microprocessors like the Pentium microprocessors use this technology.**
- Supercomputers are large computers used mainly in Scientific and Research organisations for processing complex scientific data that require huge processing power.
- **The Chinese supercomputer Tianhe-2 is at present (as on 2014) the world's fastest supercomputer, operating at a speed of $34x10^{15}$ calculations per second**
- Mainframe computers are used in large organisations such as banks, railways, airways, etc. that need frequent access to the same information and quickly process large number of transactions on-line.
- **Minicomputers belong to a class of multi-user computers which make up the middle range of the computing spectrum, in between the mainframe computers and the personal computers.**
- Microcomputer is small enough to be placed on a desk and cheap enough to be owned by an individual. Home computers (the Apple and the IBM PC), workstations, laptops etc. are examples of microcomputers
- **The first IBM PC used the Intel 8088 microprocessor and came with the MS DOS OS**
- A workstation is a high end desktop microcomputer. Workstations usually offer higher performance than a personal computer, especially with respect to graphics, processing power and multitasking abilities.
- **Laptops are portable battery operated computers and are used by people who need mobile computers in their field of work, like business executives, sales people and engineers.**
- Palmtops are small battery operated computers, that can fit inside the palm of a person and are also called Personal Digital Assistants (or PDAs)
- **An analogue computer is a form of computer that uses a particular physical property, like the electrical or mechanical characteristics of a device to solve a problem**
- A digital device uses discrete electrical voltage levels to encode a real life situation or process.

Review Questions

Q1. Fill in the blanks: 1 each

a. Pascal's Adding Machine also called the _____

b. The _____ probably is the oldest counting machine.

c. _____ was the first stored program electronic computer.

d. The introduction of _____ marked the beginning of the second generation of computers.

e. An entire CPU fabricated on a single silicon chip is called a _____

f. ULSI stands for _____

g. _____ are large computers used mainly in Scientific and Research organisations for processing complex scientific data that require huge processing power.

h. A _____ device uses discrete electrical voltage levels to encode a real life situation or process.

Q2. Multiple Choice Questions. Select from any one of the four options. 1 each

i) Which one of the following is the oldest calculating machine?
a. Arithometer b. Abacus c. Pascaline d. Napier's bones

ii) In 1617 John Napier of Scotland invented a calculating device called the:
a. Napier's Box b. Napier's Machine c. Napier's Bones d. Napier's Calculator

iii) In 1642 Blaise Pascal of France invented a calculating machine called the:
a. Pascal's Rods b. Pascal's Device c. Pascal's Machine d. Pascaline

iv) This was the first mechanical calculator that could automatically do calculations in base-10 once the numbers were fed to it:
a. Pascaline b. Arithometer c. Trigonometer d. Calculator

v) In 1823 a calculating machine called the Difference Engine was developed by:
a. Christ Babbage b. Charlie Babbage c. Charles Babbage d. Christopher Babbage

vi) The punched card reader was developed by the American inventor:
a. Herculo Hollerith b. Herman Hollerith c. Herbert Hollerith d. Hershel Hollerith

vii) The full form of ENIAC is:
a. Electronic Numerical Integrator Centre b. Electronic Numerical Integrator and Computer
c. Electronic Number Input and Calculation d. Electrically Number Input and Calculation

viii) The full form of EDVAC is:
a. Electronic Discrete Variable Automatic Computer
b. Electrical Discrete Variable Automatic Calculation
c. Electronic Data Variable Automatic Computer
d. Electrical Discrete Variable Automatic Computer

ix) The full form EDSAC is:
a. Electronic Delay Storage Automatic Computer
b. Electronic Delay Storage and Arithmetic Calculations
c. Electronic Data Storage Automatic Calculator
d. Electronic Delay Storage Automatic Calculator

x) VLSI stands for:
a. Very Large Scale Integration b. Very Low Scale Integration
c. Very Limited Scale Integration d. Visually Linear Scale Integration

xi) Built in 1990, India's first supercomputer was:
a. Param 1000 b. Param 4000 c. Param 8000 d. Param 12000

xii) The "five-function computer" was based on the concept called:
a. von Hoffmann Architecture b. von Seamann Architecture
c. von Oldmann Architecture d. von Neumann Architecture

xiii) The first electro-mechanical computer developed at the Harvard University at USA was called the:
a. Harvard–1 b. Howard–1 c. Harvard Mark–1 d. Aikeen–1

xiv) The second generation of computers used the:
a. microprocessor b. vacuum tubes c. IC d. transistor

xv) Engineers with Texas Instruments at the USA developed the:
a. Transistor b. Integrated Circuit c. Microprocessor d. Personal Computer

xvi) Computers used mainly in Scientific and Research organisations for processing complex scientific data that require huge processing power are called:
a. Workstations b. Mainframe Computer c. Personal Computer d. Super Computer

xvii) Computers used in large organisations such as banks, railways, airways, etc. that need frequent access to the same information:
a. Mainframes b. Super Computers c. Workstations d. Personal Computers

xviii) The first IBM PC used the:
a. 8085 µprocessor b. 8086 µprocessor c. 8088 µprocessor d. 8286 µprocessor

xix) Multimedia systems require:
a. Speaker b. Web Camera c. Scanner d. All of these

xx) The word CBT stands for:
a. Computer Based Training b. Computer Based Technology
c. Computer Beginners Training d. Computer Beginners Tools

SA

Q3. Short Answer type questions: 1 each

i) Name any two mechanical devices used during the early days of computing.
ii) Name the component used in the first generation of computers.
iii) Name the first fully electronic computer.
iv) State one difference between an analogue and a digital computer.
v) State one disadvantage of first generation of computers.
vi) State one characteristic of second generation of computers.
vii) State one characteristic of third generation of computers.
viii) Which generation of computers used the microprocessor?
ix) State one difference between a calculator and a computer.
x) State one use of computers in school.
xi) State the full form of DTP.
xii) Write the full form of ULSI.
xiii) Name any two components used in multimedia.
xiv) Cray is an example of what type of computer?
xv) Name the two developments in computers done by Charles Babbage.

LA

Q4. Long Answer type questions: 7 each

i) What do you mean by the von Neumann concept? State and explain the five components of von Neumann architecture. 2+5

ii) Write a short note on the first generation of computers. State two advantages of third generation of computers over second generation of computers. Name the main component of fourth generation of computers. 4+2+1

iii) State two differences between a calculator and a computer. State three differences between analogue and digital computers. State two differences between a personal computer and a super computer. 2+3+2

iv) Briefly explain the uses of computers in three different fields in modern day life. Write the name of a supercomputer. 2+2+2+1

v) Briefly explain the term multimedia. Explain the use of multimedia in any two different fields. What do you mean by the term streaming with respect to multimedia? 2+2+2+1

CHAPTER 2
Data Processing and Computer Organisation

- **Data and Information** 2-1
- **Computer System** 2-3
- **Input Devices** 2-5
- **Output Devices** 2-10
- **Memory Unit** 2-14
- **The Processing Unit** 2-17
- **Storage Devices** 2-19
- **Communication Bus** 2-25

2.1 Data and Information

In our daily life we are constantly coming around various types of data from newspapers, television, research papers in scientific journals and magazines, the Internet etc. The word **Data** (singular datum) basically means a **collection of facts or raw materials**. Examples of data are:

Data

- Marks obtained in computer studies in a class
- Daily temperatures recorded in a city over a period of time
- Cricket scores
- Customer names and addresses in a departmental store

When this raw data is processed as per the requirement of a person or an organisation, we get a more **useful and meaningful interpretation of the data which is called Information** (though there is no such sharp distinction between the two, and can be often used interchangeably). The process of converting data to information is called **Data Processing**. It includes proper storage, classification and calculations based on the form in which the data is required, or is useful.

Information

Data basically means a collection of facts or raw materials

For example cricket scores provide us with numerous cricket statistics like comparative studies of batting figures, bowling figures, individual scores etc. Distribution of black holes helps the scientists to predict the fate of the universe from the mass distribution data. Infrared photographs of the solar corona are useful in predicting solar activity, which in turn is used to predict climatic changes on earth.

When raw data is processed, we get a more useful and meaningful interpretation of the data which is called Information

Difference between Data and Information

Data	Information
Represent the raw material on which the computer works	These represent the processed raw material or data.
Data is the starting material of data processing.	Information is the end product of data processing.
The data in a system can be the information from another system. For example, the number 02091971 can be the data for a system which processes this value to get the date of birth of a person as 02-09-1971	Information processed by one system can serve as the data for another. For example the date of birth 02-09-1971 generated by a system can act as data for a system calculating the present age of a person
Example of data: The value 15042009 by itself is meaningless and serves as the input raw data for a system.	Example of information: When processed, the same value can represent a date (15-04-2009) or population of a country (15042009 persons)

The various qualities of information are:

Qualities of Information

a. **Meaningful**: Information must be meaningful and not vague in nature. Therefore the information that sea levels will rise by 1 meter for every degree rise in global average temperature, as an effect of global warming is a better information than the fact that 'ice will melt as a result of global warming'.

b. **Contain Surprise Element**: Information should have some surprise element i.e. should give some new concepts. Therefore, the prediction that a particular year will receive more than average rainfall is better information than the information 'sun rises in the east'.

c. **Reinforce Existing Knowledge**: The new information should reinforce or add to an already existing knowledge. Therefore the findings after the recent journey of the spacecraft Chandrayan to the Moon has reinforced the existing knowledge that the Moon has water under its surface in the form of ice.

d. **Rectify Misconceptions**: The new information can also be used to rectify any previous wrong information. For example the present accepted number of solar planets as 8 is a rectification over the previously accepted number 9, after the planet Pluto was declared a planetoid and not a planet.

e. **Brevity:** Information should be brief and to the point and not be very elaborate. Thus information in the form of a relation or graph (like pie-chart or a bar-graph) is better than five pages of information.

f. **Accurate**: The information should have proper accuracy. Inaccurate information is of no use.

g. **Timely**: Information should be relevant and timely and should not be backdated.

Information at one level can be data for another level. For example the data of birth information 02/09/1971 at one level may be treated as raw data at another level to calculate the present age of a person. Based on this fact, information can be classified into **five different levels** as stated below:

Levels of Information

1. **International Information**: These are information with an **international significance**. Examples include information related to the weather pattern to determine the economic effects of global warming, prices of stock in the international stock market etc.
2. **National Information**: This type of information has a **national importance**. For example information related to the economy of the country, the per capita income of the country, the management of natural resources etc. are various types of information of national interest.
3. **Corporate Information**: Such type of information is **related to a business or corporate body** and involves various departments and individuals. It includes information related to a company's profit and loss, the quarterly sales report of the company etc.
4. **Departmental Information**: This type of information is **related to a particular department** in an organisation. For example the information related to the salary structures of different individuals of a particular department forms such type of information.
5. **Individual Information**: Information **related to an individual person**. The personal record of an employee like the employee name, address, phone number, salary etc. form individual information.

Entity set → File → Records → Fields / Attributes

By nature, data can be **numeric** (i.e. numbers) or **alphanumeric** (i.e. alphabets and digits). Numeric values can be either signed whole numbers or **integers** (like -45, +80 etc.) or **floating point numbers** (like 6.023×10^{23} or 0.6023E24, +0.75E-3 etc.). Alphanumeric data can include names, addresses etc.

Physical Data indicates the process in which the raw data gets stored in the computer.

Data forms the raw material for a data processing system. The computer stores single pieces of information in units called **fields** or **attributes** like ID, name, class, etc. Several such related fields are grouped together to form **records**. The data stored in the ID, name, class, etc. fields of a particular student in a school when taken together form the student record. Several similar records like all the student records of a particular school form an **entity set**. Such a set of records which form an entity set are taken together and stored as a **file**. Within a given file a special field called the **record key** or **primary key** (e.g. ID number of a student) is used to uniquely identify a particular record.

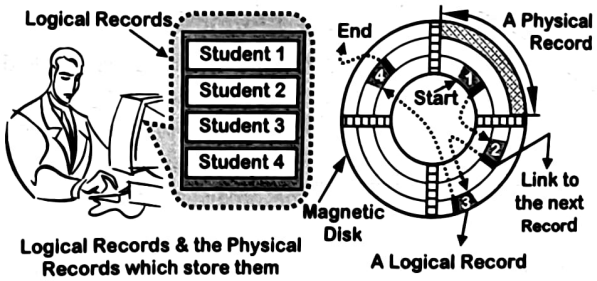
The manner in which the above data is stored and viewed/used is different. Based on the above fact we can classify data broadly into **Physical Data** type and **Logical Data** type as discussed below.

Physical and Logical Data

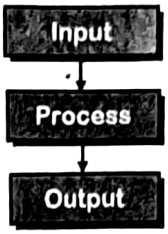
1. **Physical Data**: This indicates the process in which the **raw data gets stored** in the computer. The computer uses various types of data structures (like linked lists, trees etc.), records (like fixed or variable length records) and file organisation methods (like serial, sequential, indexed or direct files) to save the data in a physical media like a magnetic disk or magnetic tape etc.

Logical Data indicates the logical arrangement of data as viewed by the user or the programmer

2. **Logical Data**: This indicates the logical arrangement of **data as viewed by the user** or the application programmer. The main components of logical data are **entity**, **attribute** and **relationships** between the entities. For example in a school, the various entities involved are **students**, **teachers**, **subjects** taught, **marks** obtained in various examinations by students etc. The attributes involved can be the ***name***, ***class***, ***section***, ***roll*** no. of students etc. And the relationships involved can be the relationship between a student and a teacher, that between students and marks etc. All these taken together form the logical concept of data.



Data processing by a computer **involves generally three basic steps** as shown to the right, and the total process is called a **Data Processing Cycle** or **IPO cycle.**



IPO Cycle

1) **Input**: During this phase the raw data to be processed is input into the computer system by means of various input devices like keyboard, mouse, scanner etc. and stored in various media like the hard disk, RAM etc. before it is processed.

2) **Process**: This phase is used to actually modify or process the data to get the desired information. This phase can again involve various operations like:

   a) **Record**: This involves transfer and storing of the input data for use during the processing.

   b) **Duplicate**: This involves making copies of the input data for multiple uses. For example, duplication of the bill against a purchase may be required for the customer and the retailer.

   c) **Verify**: The recorded data is checked for any errors before it is processed.

   d) **Classify**: The input data may need to be classified into various categories for proper processing. For example the student data can be classified into various classes and sections.

   e) **Sort**: Processing may also involve sorting the data in a logical sequence. For example student records may be sorted in terms of class and section for convenient viewing.

   f) **Merge**: Often two or more sets of data may need to be merged together. For example data from various sales counters may need to be merged together into a single transaction file.

   g) **Calculate**: This involves various numerical operations that are done on the input data.

   h) **Search**: This operation is used to look for particular information from the total data set.

3) **Output**: The processed data needs to be output and returned back to the user. This output can either be displayed to the user, taken in a printed form, or can be stored for further processing.

Data processing by a computer involves Input, Process, and Output and the total process is called a Data Processing Cycle or IPO Cycle

Various **data processing systems** have evolved over the years to process data. Some of these include:

Data Processing Systems

1. **Management Information System (MIS)**: This involves data processing related to the internal working of a business for proper business management. It deals with solving business related issues like product costing, creating business strategies, managing human resources etc.

2. **Decision Support System (DSS)**: This is a computerised data processing system that supports a MIS and deals with business decision-making activities. A properly-designed DSS is an interactive software-based system which helps decision makers in an organisation to compile useful information from various raw data and personal knowledge. The information so generated is used to identify business problems and make decisions to solve them.

3. **Electronic Data Processing System (EDPS)**: Computers are used to process data electronically. The process is fast and accurate and the data can be stored and retrieved later.

4. **Transaction Processing System (TPS)**: Such a system is used to process data during a transaction in database system. A transaction program is used to maintain data consistency during a transaction. For example during an electronic payment the amount must be both withdrawn from one account and added to the other. In case a complete transaction cannot take place, the partially executed transaction must be 'rolled back' to the original state.

2.2 Computer System

Computer System

In the previous section we had seen that a modern day person processes data by using various data processing systems, the backbone of which is an electronic computing machine or computer. Let us now discuss the various features that a computer should have in general to process data.

1. **Solve all types of arithmetic and logical calculations**: The computer must be able to carry out all types of arithmetic operations like addition, subtraction, multiplication and division and at the same time do logical comparisons between various values. The Arithmetic and Logic Unit (ALU) of a modern day computer does this job.

2. **Must have a storage unit**: The computer should have a memory unit to store various types of information. This information can be the raw data to process, the instructions to process this data and the processed data itself. In a computer the RAM and the hard disk serve this purpose of storing data temporarily or permanently.

One of the world's leading computer and computer peripheral manufacturer, Hewlett Packard (HP) was first started in a garage at Palo Alto, in 1939

3. **The computer must have input/output devices**: Unless we input data to a computer we will not be able to process it. For this we need special input devices. Similarly the processed data should be available to the user. This is done through the output devices in a computer. **Keyboards**, mouse etc. serve as input devices and **VDU** (video display unit) printers etc. work as output devices.

4. **The computer must be able to decide**: The computer must be able to decide on the course of action it should take based on various conditions. For example if the computer has to print a number as an even number or as an odd number based on the value of the number, then it should be able to take that decision. It should be able to decide upon the condition and print the appropriate result. This decision making capability is what separates a computer from a ca lculator. The **control unit (CU)** of the computer is responsible for carrying out this function.

The basic physical building blocks of a computer are collectively called Hardware

From the above discussions we have seen that for data processing we require an **input** device, a **processing** device, memory, and finally an **output** device. These form the basic building blocks of a computer and are collectively known as computer **hardware**. Hardware consists mainly of the CPU, memory modules, input/output devices like keyboard, VDU, etc. and storage devices like hard disks etc.

A sequence of instructions given to a computer to make the hardware work, in a language that the computer understands, is known as Software

However computer hardware cannot do anything on its own and it has to be instructed to do any desired job. **A sequence of instructions** needs to be given to the computer in a language that the computer understands to make the hardware work. Such a set of instructions is known as **software**. Software basically consists of **programs**, which tell the computer what to do and how to do the processing. Programs usually contain a set of instructions in coded form, written to serve a specific job or purpose. Without the software the computer would be a dumb terminal.



Firmware

There can be two different types of software based on how these are stored. In case a part of the software is **permanently stored in a computer** in a read only type memory, then such software is called **firmware**. Firmware basically keeps information related to the computer system.

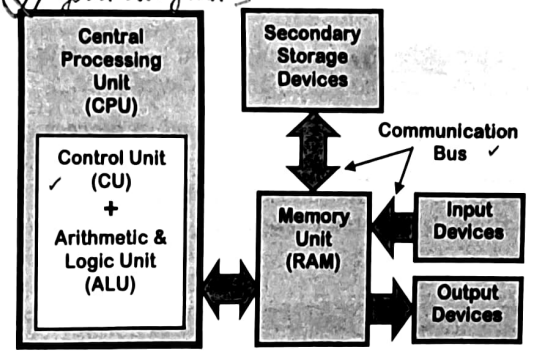
In case software is permanently stored in a computer in a read only type memory, then such software is called Firmware

Apart from the computer hardware and software, there is a third entity without which the computer will not run. It is called **liveware or humanware**. It basically consists of people who work with the computer and consists of programmers, operators, system analysts, end users and others.

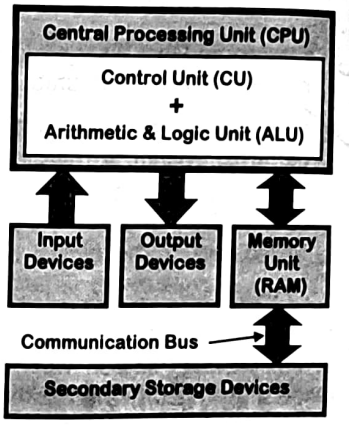
As stated earlier, the **hardware** of a computer i.e. the physical components of a computer can be subdivided into five basic parts as shown in the next page.



The above hardware is **connected to each other in a logical manner** to carry out the process of input, output, processing and storage of data. **Block diagram-1**, shows how these different parts are generally connected together in a computer.



Block Diagram-1 of a General Computer



Block Diagram-2 of a General Computer

The computer mouse, the windowing graphical user interface, laser printing and the network card were all developed at Xerox Corp. in Palo Alto, California

Input devices like keyboard, mouse, scanner etc. input the data from the user and send them to the **memory unit** (the RAM). The **central processing unit** (CPU) then fetches the data from the memory. The **control unit** (CU) and the **arithmetic and logic unit** (ALU) then process the data and send the result back to the memory unit. Next the processed data is either stored in **storage devices** like the hard disk and CD ROMS, or it is passed to the **output devices** like the monitor, printer, or speaker. The **communication buses** like the data, address, and control bus are responsible for carrying information from one part to another.

In case the input/output devices **directly communicate with the CPU**, then the **block diagram-2** can be used to describe the working of such a general purpose computer.

2.3 Input Devices

Input Devices help to **interact with the computer and accept data** and instructions from the user. Depending upon the various types of data entered like textual data, images, audio and video data etc. let us now discuss the various types of input devices available.

- Keyboard

The **most common input device is the Keyboard** which is usually connected to the computer using a **PS/2** or an **USB port.** It resembles a mechanical typewriter where the alphabets are placed in the **QWERTY format** along with the numerals 0 to 9 and punctuation symbols. It is **used to enter textual data.**



Keyboard

The number of keys on a keyboard varies from the original standard of **101 keys to the 104-keys** as in Windows keyboards. These keys can be divided into the following sets based on their functions:

Key Set	Function
Alphabet Keys	Used to type the alphabets A to Z in lower and upper case
Digit Keys	Used to type the digits from 0 to 9
Punctuation and Symbol Keys	Used to type various punctuations (like , .' " ! etc.) and symbols like &, *, +, %, { }, ( ), [ ], blank key (spacebar) etc.
Modifier Keys	These are a set of keys which when pressed along with some other keys, modify the function of that key. These include the **Shift** key, **Alt** key, and the **Ctrl** key. For example when we press the key 'A' on the keyboard it prints 'a' in lower case, but when we keep the Shift key pressed and then press the 'A' key, then the upper case 'A' gets printed
Lock Keys	There is another set of keys on the keyboard which have an ON and an OFF state. The **Caps Lock**, **Num** Lock and **Scroll** Lock are examples of such keys
Navigation Keys	These include the 4 **arrow** keys, **Page up**, **Page down**, **End** & **Home** keys
Edit Keys	Used to edit a portion of a file and include Enter, Back Space, Insert, Delete, and Tab keys
Function keys	These include the **F1**, **F2**, ... **F12** keys and are used to perform some special work as specified by the operating system when these are pressed. For example the Help file related to the current program running on the computer opens when you press the F1 key
Multimedia Keys	Nowadays multimedia keyboards come with some special buttons which can be used to enable or disable sound, open a particular home page, open the explorer etc.
System Keys	These contain the **Break**, **Esc**, **Print Screen** and similar keys

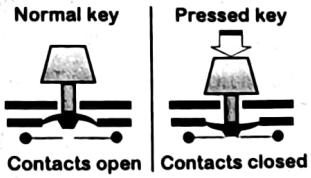
The IBM ThinkPad's butterfly keyboard enables a standard-sized keyboard to be put inside a smaller sized laptop for more comfortable typing. The keyboard, formed of two joint blocks, expand and withdraw when you open and close the laptop display

On a QWERTY keyboard, Alaska is the only country that can be typed using only one row of keys

A keyboard has it own processor and circuitry to carry information to and from the processor. The major part of this circuitry is the **key-matrix**. It is a grid of conducting lines placed beneath the keys and used to detect a key press. Depending upon the technology used, a keyboard can be mechanical or capacitive in nature.

The **internal working mechanism** of a common mechanical keyboard is discussed below:

- When a key is pressed on the keyboard, it pushes a rubber dome beneath the key. A conductive contact on the underside of the dome **touches a pair of conducting lines** on the circuit below.
- This **closes the contact** between the lines, causing a change in the current flowing through the lines.
- A special signal called a **scanning signal** detects this change of current and the keyboard chip generates a **scan code** signal that corresponds to the key pressed.
- The scan code generated is then **sent to the computer's basic input output system (the BIOS)** via the keyboard cable. The BIOS inside the computer receives the scan code and **decodes it** into a corresponding ASCII value. American Standard Code for Information Interchange
- The computer then **decides what action to take** on the basis of the key pressed.



Working of a mechanical Keyboard

• Mouse



An early wooden mouse

A Mouse is a point and click input device used to point and select any item on a GUI

After the keyboard, the **mouse** is the next most useful input device. It is a point and click **input device which is used to point and select any item** on a **Graphical User Interface (GUI)**. It is a small object which looks a bit like a mouse and can be moved along a surface. It consists of **one or more buttons which can be clicked** to do various functions. The user can **simply point to options and icons** on a GUI screen and select them by clicking the mouse button.

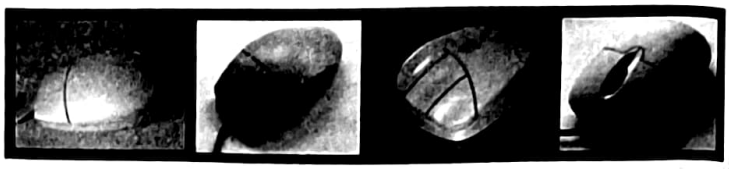
A mouse can be **wired, or wireless**. Wired mouse can be connected through wires, which are generally connected through basically a **PS/2 or an USB port**. Wireless mouse use Bluetooth technology to communicate with the computer. These use a Bluetooth transmitter to send signals, which are then received by a Bluetooth transmitter. The transmitter is attached as a small dongle to an USB port in the computer.

Generally nowadays a mouse comes with **three buttons, along with a scroll wheel**. Each button has its own function. Usually when the **left button** is pressed an item is selected, and when it is rapidly clicked twice then an item is opened. The **rightmost button** when pressed shows a drop-down menu. The scroll wheel can be rotated upwards or downwards to scroll through a long document or a webpage. The **scroll wheel** when pressed acts as the **middle button**. The middle key also has some specific features. Like in Linux environment a middle click generally pastes the latest content of the clipboard.

The first mouse, invented by Doug Engelbart in the year 1964, was made of wood

The first computer mouse was introduced to the market in 1968 at an exhibition

When the mouse is moved, a **graphic arrow head called the mouse pointer** moves on the screen. Using this pointer, any object on the screen can be pointed, clicked and selected. Once selected, an item can be dragged and dropped from one place to another, or can be opened.



One Button, Two Button, Three Button and Scroll Mouse

Based on the way a mouse determines its position on a surface, we have broadly **two types of mouse**, a mechanical mouse and an optical mouse. A **mechanical mouse** uses a ball-and-roller arrangement to determine the position of the mouse pointer. An **optical mouse** on the other hand uses an LED light source and a sensor to scan the surface where the mouse is placed and determine the position of the mouse pointer on the screen. Modern optical mouse use invisible IR Laser light as the light source. These are more accurate than the LED mouse and can be used over any surface.



Scanner

• Scanner

A scanner is an **input device** which can be **used to input images and documents** and store them in a digital format in the computer. The different types of scanners are:

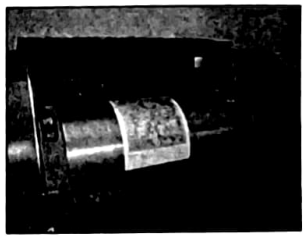


1. **Flatbed scanners:** In these types of scanners the document is placed on a glass bed underneath which the scanner head moves as it scans. The scan head is moved with a rotating belt. The picture on the right shows such a scanner.
2. **Sheet-fed scanners:** These scanners have an immobile scan head. In this case the paper document is fed into a roller which rolls the paper in and makes the whole document travel under the fixed scanner head.

A Scanner is used to convert an image or a document to digital format for storing in a computer



3. **Handheld scanners:** Similar to flatbed scanners but the scan-head needs to be moved manually on the document in order to scan it. This type of scanner typically does not provide good image quality. However, it can be useful for quickly capturing text or image.



4. **Drum scanners:** These are used by the publishing industry and can scan highly detailed images. These use the photomultiplier tube to capture the image. The **document to be scanned is mounted on a glass cylinder.** A sensor in the centre of the cylinder splits light which is reflected from the document into three beams. Each beam is sent through a colour filter to a sensor, where the light is changed into an electrical signal.

The **flatbed scanner is the most widely used** one. To scan a document it is first loaded on the glass plate of the flat bed scanner. The lamp inside the scanner illuminates the document. The scan head then moves across the document. As the lamp illuminates the page, the light from the document is reflected off onto a lens. The lens focuses the image on a special sensor called Charged Coupled Device (CCD) array. The CCD

converts the light signal to electrical pulses. The light is then converted to raw digital data and sent to the computer for proper storage.

• Barcode Reader

A barcode reader is **used to scan ready product information from product labels.** It is used in supermarkets, shops, libraries, post offices and other places. The information on the barcode usually contains the product ID, price, name of the manufacturer, etc. printed on the labels.

A standard barcode **consists of a number of parallel light and dark bars of variable width**. The code is read optically using a special scanner. The barcode is designed **based on a 10-digit number system**. Of these, 5 digits are used to encode information related to the manufacturer and the remaining 5 digits are used to encode the particular product details. The code numbers are also written under each bar symbol which can be used in the event of a machine failure.





Barcode Reader

Barcode Reader is used to scan product information from special coded labels

Two types of barcode systems are available. These are the **Uniform Product Code (UPC)** used in the US, and the **European Article Number (EAN)** used in Europe.

• Touch Screen



A touch screen is a special display screen where the **user can enter data by touching the screen** at specific locations. It is a display which can detect the presence and location of a touch within the display area. It was developed in the second half of the 1960s and is widely **used in applications like Personal Digital Assistant (PDA)**, mobile phones, interactive railway or airlines enquiry systems, bank ATMs etc.

Using a touch screen, the **user can directly interact** with what is displayed on the screen, rather than indirectly with a mouse. It **allows one to use his finger** or hand without requiring any intermediate device like a stylus that needs to be held in the hand.

Touch Screen

In a **TouchScreen** the user can input data by touching the screen only

Various types like **resistive, capacitive, or infra red** touch screens are available. The resistive touch screen is presently the **most widely used** and affordable touch screen technology type as it is not affected by heat, dust or water.

• Light Pen



A **light pen is a point and draw input device** which allows the user to point to objects displayed on a CRT screen. These are also used for computer aided design applications, where the **user can directly draw on the screen with the help of a light pen**. It is similar to a touch screen but has a greater accuracy.

Light Pen

A **Light Pen** is a point and draw input device which allows the user to point to objects displayed on a CRT screen

It is basically a **light-sensitive device that uses a photoelectric cell and an optical lens** mounted in a pen shaped case. It needs to be used along with the computer's CRT monitor. The pen is held by the hand and directly pointed to the screen to select menu items or icons or directly draw graphics on the screen. The pen has a finger-operated button, which can be used to select options from the screen by clicking on it.

The pen **works by sensing the** light signal from the CRT monitor screen. It converts the light into electrical pulses and sends them to the processor. Analysing this light signal the (x, y) position of the pen is found out by the processor. The signal then identifies the item that is triggering the photocell on the light pen and proper action is taken.

• Optical Mark Reader (OMR)

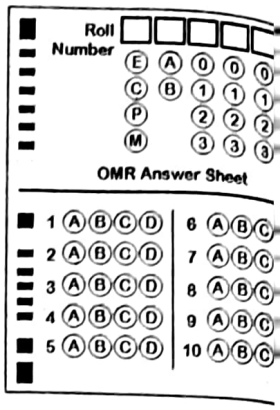
Optical Mark Recognition (also called Optical Mark Reading or OMR) is the **process of detecting data from pre-printed document-forms marked by humans**. One of the most familiar applications of optical mark recognition is OMR answer sheets in multiple choice question examinations.

Optical Mark Reader

A special kind of sheet/form is used for this purpose. The user has to **input information by darkening circles** marked on this pre-printed sheet using a HB pencil (picture of a sample OMR form is given to the right). The sheet is then automatically read by a scanning machine which shines a beam of light onto the

OMR or *Optical Mark Recognition is the process of detecting data from pre-printed document-forms marked by humans*

OMR form. **The contrasting reflectivity from the marked and unmarked areas** is then used to detect the marked areas as they reflect less light than the blank areas of the paper.



Desktop OMR software is available which allows a user to create forms in a word processor and print it on a laser printer. The OMR software then works with a common desktop image scanner to process the filled forms.

The process of OMR is however **not suitable for gathering large amounts of text**. There is also the possibility of missing data during the scanning process. However for all practical reasons OMR provides a fast, accurate way to collect and input data.

Optical Character Reader

- Optical Character Reader (OCR)

Optical Character Recognition (**OCR**) technology is basically used to **optically scan pages containing t** and then to identify and **save the scanned data as a text file** instead of an image file. The image is sto in the computer's memory as a bitmap image consisting of a grid of dots. The **character recogniti software** of the scanner then converts the array of dots to equivalent ASCII text, which the computer interpret as letters, numbers or special characters. Intelligent systems with high **recognition accuracy** most fonts are now available. Some systems are even capable of reproducing formatted output that clo approximates the original scanned page.

OCR *is basically used to optically scan pages containing text and then to identify and save the scanned data as a text file*

The advantages of using OCR technology are given below.

- **Reduced Storage Area**: The storage area required by a text file is several times smaller than that of image file.
- **Word Processing**: Since the scanned document is converted and saved as a text file, word process of the document can be easily done.
- **Quick Data Evaluation**: One can use OCR to evaluate and tabulate large amount of data quickly scanning specially marked sheets as in specially marked entry forms.

Magnetic Ink Character Reader

- Magnetic Ink Character Reader (MICR)

Magnetic Ink Character Recognition or **MICR** technology is **used mainly by banking systems for fas processing of large volumes of cheques**. Banks that use MICR use a special type of cheque. The ban identification code, cheque number and account number are pre-printed using a **special magnetic ink t** contains particles of iron oxide. Magnetic printing is used so that characters can be easily read, even wh they have been overwritten with other marks.

MICR *is used mainly by banking systems for faster processing of large volumes of cheques*

The **characters are also of special font**. The major MICR fonts used around the world are **E-13B** and **CMC-7**. Almost all Indian, US, Canadian and UK cheques now include MICR characters at the bottom of the paper in the E-13B font (shown to the right).



When a customer deposits a cheque with the bank, a **MICR reader is used to recognise the magne ink characters automatically** and carry out the transaction. When the cheques enter the reading unit, th pass through a magnetic field, which magnetises the particles of ink. Read heads then examine the characters by examining their shapes. The **special shapes of the MICR font ensure that each let produces a unique field** for the character recognition system to provide a reliable result.

Difference b/w OCR & MICR

Differences between OCR and MICR

OCR	MICR
Used for reading any type of printed text material	Used in banks for reading special information from cheque
The fonts scanned usually can be of a variety of shapes.	The fonts scanned should be of a pre-defined shape like th E-13B font.
The ink in the text document scanned can be of any type.	The ink used in MICR is a magnetic ink made using particl of iron oxide.
The scanned document is usually stored as a text file for editing or printing.	Information from the scanned data is used for processing cheques in a bank.

Web Camera



Web cameras or simply web cams are **input devices used for video capturing** and are connected to a computer through an USB port. Presently computer hardware manufacturers are manufacturing **laptops with built-in web cams**. This eliminates the need to connect a web cam through an external USB port.

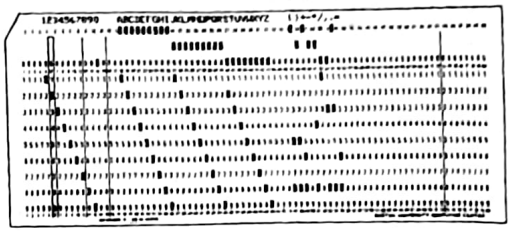
Web cams can be **used as web-accessible cameras** to do personal video-conferencing using instant messaging text chat services such as AOL Instant Messenger. Increased video quality in modern web cams has helped PC users worldwide to do **one-to-one live video communication** over the internet. Programs like **Yahoo Messenger**, AOL Instant Messenger (AIM), Windows Live Messenger, Skype etc. offer videoconferencing support.

Today web cams are also used to provide views into offices and other buildings for security purposes, monitor traffic and do a variety of other jobs.

Web Cam

A **Web Camera** or simply a web cam is an input device used for video capturing

Punched Card Readers and Tape Readers

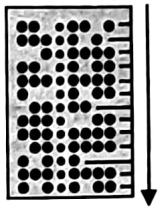


Card readers and tape readers were used during the early days of computing **to input data that were punched in special types of paper cards** and ribbons. These machines are obsolete now.

Card readers are **electromechanical devices** that are used to read the information punched into a card. The **presence or absence of a hole** in the card is converted to an electrical signal representing a binary '0' or a '1'. Each card (picture shown above) usually consists of **12 rows and 80 columns** and are 3¼" wide and 7¾" long. The most commonly used code to encode data into the card is the **Hollerith Code**, where a single character is punched in each column of the card.

Punched Card

The punching machine contains a hopper where the blank cards are first stacked. The operator then **uses a keyboard to enter the data** which is then punched into the card by proper electromechanical mechanisms. To read data from the card, the card is moved over a row of **80 read brushes**. If a hole is present, a contact is made by the brush. This produces a signal that can be used by the computer to interpret a '0' or a '1'.



Punched tape readers were used to **input information punched on a paper tape**. Like card readers, these are **electro-mechanical devices** where mechanical sensing pins are used to read the character punched in each line of the tape. The **presence and absence of a punch or a hole in the tape is used to encode a binary '0' or a '1'**. Faster tape readers have photo electric cells to detect a hole in the tape.

Punched Tape

The width of a tape varies from ½" to 3". Information is punched into a tape a line at a time. A single character is punched as a pattern of holes in each lateral line. When a key in the tape punch keyboard is pressed, the **binary coded symbol of the character is punched** into the tape and the tape then advances to the next line. Many tape punches are able to read from a perforated tape and then produce a printed copy of the code in the tape.

Other Input Devices

Apart from these there are a variety of input devices available presently. They consist of **Trackballs, Graphic Tablets, Joysticks, Microphones, Digital Cameras**, etc.

A **trackball** is similar to a mouse and used as a pointing device in laptop computers. A laptop computer can also have a **touch pad device** over which the user can move his finger to move a pointer similar to a mouse pointer on the screen.



A **graphic tablet** or simply a tablet, (picture shown on the right) is an input device on which the user can draw anything using a special pen and the image is directly transferred to the computer display and can be saved.

A **joystick** is another type of input device **mainly used in computer games** and simulation software to input a move. These work as navigation devices and can have special buttons to input various actions required in a computer game.

Microphones can be used to directly input audio signals like voice or music into the computer with the help of specific software. **Digital cameras** can be used to **take an image in a digital format** and directly transfer to a computer as a digital image file. This saves the need to print and scan an image.

2.4 Output Devices

The function of an **Output Device** is to present processed data to the user. Different types of output devices are available for this purpose. These include various types of video display units like **CRT and LCD monitors**, various types of **printers** like dot matrix, inkjet, and laser printers, **Speakers,** etc. Each of these is discussed below.

A VDU displays a picture on a screen based on the video signal sent by the computer

- Video Display Units



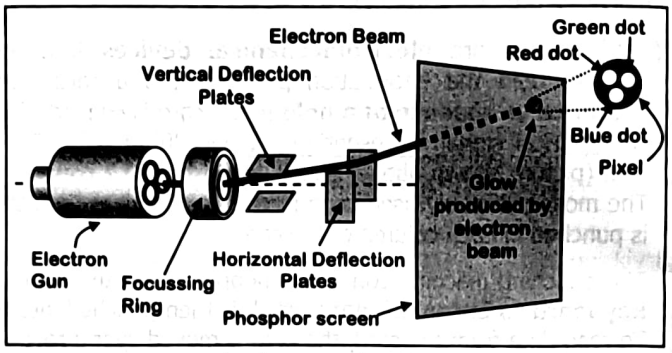
A visual display unit (VDU) or commonly a **video monitor is the most widely used output device**. It is made of a **display screen** and **electronic circuitry** that displays a picture corresponding to the video signal sent by the computer. However it **does not produce a permanent record** of the output. There are **two main technologies** used to manufacture a VDU, namely **CRT** (picture on the right) and **LCD** type displays, as described below.



CRT Monitor

CRT Monitor

The basic components of a Cathode Ray Tube (CRT) monitor are the electron gun, focussing systems, horizontal and vertical deflection systems, and a phosphor coated screen. The phosphor material is arranged into an array of millions of tiny cells, usually called dots.



Working of a CRT Monitor

A **colour monitor has three electron guns** that correspond to the primary colours red, green and blue and emit streams of electrons very close to each other causing the beams to overlap. The human eye views the three overlapping dots created by the electron beams as a single dot. A **cluster of 3 dots for the colours red, green and blue together form a pixel**. It is the smallest unit of a picture element on the screen. The colour and brightness of a pixel is controlled by adjusting the intensities of the electron guns. The **resolution of a screen depends on how close the pixels are placed**.

A cluster of 3 dots for the colours red, green and blue together form a Pixel on a VDU screen

Each electron beam scans across the top of the monitor **from left to right**. After reaching the right edge of the screen, the beam is blanked and moved back to the left edge at a position slightly below the previous trace. The scan continues until the bottom right corner of the screen is reached. In this way the electron gun **redraws the entire image**. ← redrawing image

A **display adapter** within the computer is used to generate the video format. The image to be displayed is processed by a dedicated processor in the display adapter and the image information is converted to pixel information. In a **high resolution colour monitor 8 bits are used per pixel** to represent different intensities of the colours red (8bits), green (8bits), and blue (8bits). Thus using 24 bits a total of 2^{24}=16.8 million colours can be produced.

One can operate a VDU either in **text mode or in graphics mode**. In text mode a monitor displays 25 lines with 80 characters per line. Thus such a screen can display 25x80=2000 characters. Depending on the output capacity, a display adapter can be a Monochrome Display Adapter **(MDA)**, Colour Graphics Adapter **(CGA)**, Video Graphics Array **(VGA)**, Super Video Graphics Array **(SVGA)** or Extended Graphics Array **(XGA)**.



LCD Monitor

LCD Monitor



Another type of display is the **flat panel display**. It is formed using special types of crystals called **liquid crystals** and hence called **Liquid Crystal Displays** (LCDs). These have **lower power consumption**, flat screens, and are thin in nature with less than 3" in depth and hence occupy lesser desk space. However the disadvantage with these monitors is a **narrower viewing angle** as compared to CRT displays.

In an LCD display, the image is produced by placing the **liquid crystal** between two glass plates. A **special arrangement of light polarisers and electrodes** are then used to form the image on the screen. The image is formed by passing light from a light source (like LED) through the liquid crystal and by changing the property of the crystal by passing electric current through the electrodes.

There are two types of LCD's namely **Passive Matrix LCD** and **Active Matrix LCD**. The first type uses a **set of electrodes** arranged in rows and columns forming a matrix to control the pixels on the screen. The second type uses a matrix of **Thin Film Transistors (TFT)** to drive the pixels. TFT LCD monitors are brighter and costlier and are also used in laptop computers.

Differences between CRT and LCD monitors

Difference b/w CRT & LCD monitors

CRT Monitor	LCD Monitor
Uses cathode ray tube to generate an image	Uses liquid crystals to generate an image
Power consumption is high	Power consumption is low
Displays are either rounded or flat	Displays are always flat
Brightness in more	Brightness in less
Viewing angle is more	Viewing angle is less
These displays are bulky in nature and occupy more space	These displays are thin in nature and occupy less space
Less expensive than LCD displays	More expensive than CRT displays
Cannot be used as displays for laptops	Can be used as laptop computer displays

Features of Video Display Units

Features of VDUs

In general any video display unit has the following features:

a. **Paging**: Using paging the monitor can switch to the next or previous page when requested.

b. **Scrolling**: Using scrolling the data/image can be moved line-by-line in an upward or downward direction. In some applications (e.g. Excel) a page can be scrolled in a horizontal direction also.

c. **Screen Position Control**: Based on the position of the cursor on the screen, the user is able to enter information at that point.

d. **Reverse Video Effect**: This is used to highlight a portion of the display screen in reverse colour.

e. **Blinking**: To draw the attention of the user, the screen cursor blinks in reverse video.

f. **Brightness Control**: To highlight some data, it can be displayed in increased brightness.

Impact Printers form an image on paper by using some electro-mechanical impact device and ink ribbon

- Printers

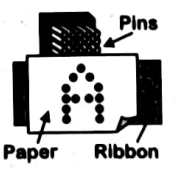
Unlike a VDU, a printer is an output device which can be **used to produce a permanent copy of the output**. There can be two broad classifications of printers – **impact** and **non-impact printers**.

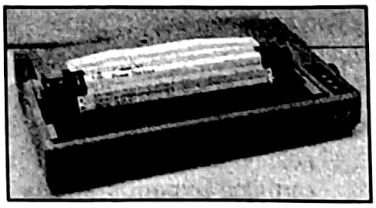
Impact Type Printers

In an impact type printer, the **image is produced on paper by some electromechanical impact device** just like the working of a mechanical typewriter. Various types of impact printers include Dot Matrix Printers, Line Printers, Daisy Wheel Printers etc. as described below.

Dot Matrix Printer

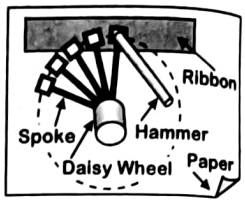
1. **Dot Matrix Printer (DMP)**: Instead of printing a character at once, these printers **use a group of pins of very small radius** to strike the ink ribbon and create a dot at the point of impact. A character or an image can be formed by proper arrangement of many such dots. The **pin hammer assembly** is moved back and forth by a motor. To print a character, driven by electromagnets the pins hit the ink ribbon and as a result a character gets printed on the paper. A roller moves the paper to the next line and continues printing. Usually a pin **head assembly consists of 7x5 dots or 9x7 dots** arranged in a matrix. Higher number of dots creates finer impressions. These printers are still used in offices for heavy printing jobs and can print for very long time without any break. Moreover any type of character sets can be printed.





Daisy Wheel Printer

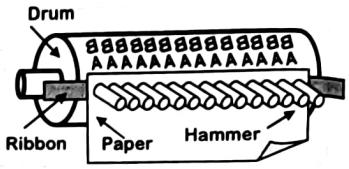
2. **Daisy Wheel Printer:** These are character printers where a special type of wheel called a Daisy Wheel contains the set of characters used in the printing process. The **characters are embossed** on small metal plates that are mounted on the spokes of a wheel (see diagram to the right). During printing, the wheel rotates at a high speed and an electro-mechanical **hammer strikes on the appropriate character** when it comes under the hammer. The character is struck against an **ink ribbon** which drives out the ink from the ribbon and makes an impression of the character's shape on the paper. By changing the wheel, one can print using a different set of characters. Such a printer is **fast and good for printing basic text**. It can print at the rate of 10 to 50 characters per second. However it prints a single character at a time and **cannot be used to print images.**



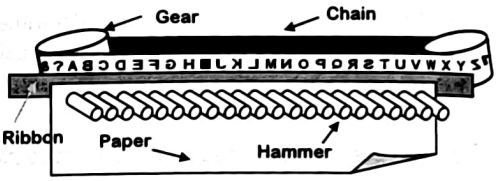
Line Printer

3. **Line Printer:** This type of impact printer is used to **print a complete line of text at a time** at a very high speed. These printers have a hammer for each column of the page which strike on an ink ribbon at the same time and a line is printed at once. These are **much faster** than those which print one character at a time. However these are **costly and heavy** as many sets of hammers are needed for printing. Two main types of line printers as described below:

a. **Drum Printer:** Such a printer contains a **cylindrical drum** whose length is same as the length of a printed line. The drum has **132 parallel tracks** and along each track all the characters of the character set are embossed. The 64 character set is the most common one. The printer ribbon and the printing paper are sandwiched between the drum and a set of 132 hammers, each corresponding to a particular character. The **drum rotates at a high speed** and when the appropriate character in a track comes under the hammer, the hammer strikes on the paper and the character impression from the drum is printed on the paper. In **a single rotation of the drum** all 132 hammers are activated and **the whole line gets printed**



b. **Chain Printer:** In such a printer, the **printing characters are chained together to form a loop**. Using a motor-gear assembly, the chain is rotated horizontally at a high speed. An ink ribbon and paper is mounted in front of the characters on the chain. A set of hammers, equal to the number of printing positions on a line are placed touching the paper. When the appropriate character comes under a hammer, the hammer strikes the paper and the character impression from the chain is transferred to the paper through the ink ribbon. The printer can print from **5 to 40 lines per second.**



In a Non-Impact type printer, the image is produced on paper without the need for any impact hammer or ink ribbon.

Non-Impact Type Printers

In a non-impact type printer, the **image is produced on paper without the need for any impact hammer or ink ribbon**. These can be used to print both character and image printouts and produce much less noise during printing. The **printing quality** of these printers is also **far better** than impact type printers and can produce both black & white and colour printouts. Moreover non-impact printers are usually **faster than the impact ones**. But the major disadvantage is that these **cannot be used to produce multiple copies** of the same document at the same time.

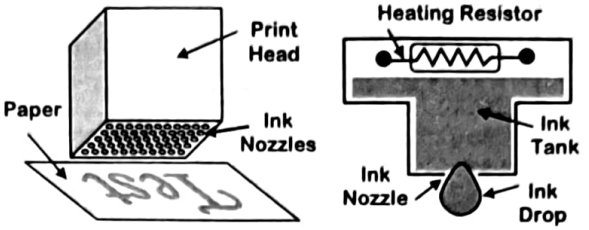
Two main non-impact printers include **Inkjet Printers** and **Laser Printers** as described below.

Inkjet Printer

1. **Inkjet Printer:** An inkjet printer is a non-impact printer. It prints by **ejecting microscopic drops of ink through tiny ink nozzles** in the print head. Each nozzle can be heated up selectively in a few micro-seconds using resistive heating effect. **On heating, the ink droplet** in the ink-tank of the ink cartridge **expands and is ejected** out from the nozzle onto the paper. This ink drop forms a tiny dot of ink on the paper. Several such dots combine to produce a particular character or image. The cartridge assembly moves sideways as it keeps ejecting ink droplets on the positions where it is instructed. A roller automatically rolls in the paper so that the printer can print the next line.



As tiny dots of ink combine to produce a pattern, they can be used to **print both photo quality images and characters** of any shape and size. A high resolution ink-jet printer can print even **2400 dots per inch**. Depending upon the number of ink cartridges used, an inkjet printer can produce both monochrome and colour output. A colour inkjet printer usually contains 4 different ink cartridges or ink tanks namely **Cyan, Magenta, Yellow and Black** (collectively called CMYK). By proper combination of these 4 different colour dots, all other colours can be created on the paper. These printers are cheap to buy and can print at speeds of 5 to 8 pages per minute. However **inkjet printers are slower than laser printers** and are not suitable for bulk volume printouts.



Working of Inkjet Printer

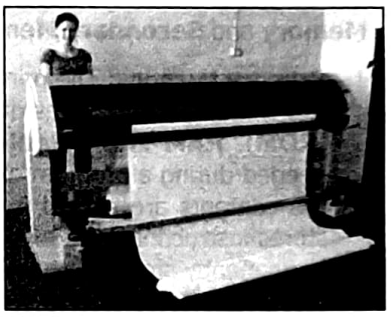
2. **Laser Printer**: These non-impact type printers use **dry ink** (called toner) to print text or images on paper. A laser beam is used to charge portions of a roller drum coated with a special photo-conducting material. The charge pattern laid down on the drum depends on the content of the document to be printed. As the paper is rolled over the drum, dry ink is then deposited onto the paper based on the static charge on the drum. The ink is then bonded onto the paper using heat.



Laser Printer

Colour laser printers have four different drum and toner assemblies corresponding to the basic colours cyan, magenta, yellow, and black. Laser printers are **costly** but can **print at high speeds** of 8 to 15 pages per minute and are **suitable for bulk printing** jobs of text and images. The output of a laser printer is also better than that of an ink-jet printer and is permanent in nature.

3. **Plotter**: A plotter is similar to an inkjet printer, but is used for **printing on larger papers or continuous paper rolls**. These are usually used for **printing engineering drawings** like building plans, mechanical drawings, assembly drawings etc. A plotter can print in **both black and white and colour** and can either have a set of **plotting pens** filled with the colours cyan, magenta, yellow and black in case of a colour plotter. Depending upon the working one can have a **drum plotter** (picture on the right) where the paper is rolled over a drum and a **flat-bed plotter** where the plotting pens plot on a flat surface over which the paper is placed.



Plotter

Comparisons between VDU and Printer

VDU	Printer
Output device, which can display text, graphics, or both	Output device which can print text, graphics, or both
A VDU can be either in black and white or in colour.	A printer can also be in black and white or in colour.
Colour VDUs have three different assemblies for the primary colours red, green and blue.	A colour printer has four different colour cartridges, cyan, magenta, yellow and black.
The output is temporary and is lost once the computer is switched off, as it is displayed on a CRT/LCD screen	As the output is usually on paper, it is permanent and can be used later.
Usually no other extra accessories are required to run a VDU.	Accessories like paper, ink-ribbon or ink cartridges are required.
A VDU is usually not provided with any memory.	Printers usually have their own buffer memory.
Various types of VDUs include CRT and LCD monitors.	Various types include Dot Matrix, Inkjet, Laser printers

Comparison b/w VDU and Printer

Differences between Impact and Non-Impact Printers

Impact Printers	Non-Impact Printers
Print head is operated electromechanically.	Print head does not have an electromechanical device
They print by hammering a set of metal pins or character sets on an ink ribbon.	These print by depositing ink drops or dry powder ink from an ink cartridge.

Comparison b/w Impact and Non-Impact Printer

Impact Printers	Non-Impact Printers
Can be used to produce multiple copies by using carbon papers.	As no impact mechanism is there these cannot be used to produce multiple copies.
Generally used to produce character outputs.	Can produce both character outputs and images.
Usually produce output of a single colour.	These can produce outputs of different colours.
Since these use electromechanical devices these are noisy to operate.	The absence of electromechanical parts makes these much less noisy to operate.
These printers are slower than non-impact printers.	These are faster than impact printers.
E.g. Dot-Matrix, Daisy-Wheel printers etc.	E.g. Ink-jet, Laser printers etc.

- Other Output Devices

Other output devices include speakers, computer output microfilms (COM), CAD-CAM machines etc. The processed audio output from a computer can be heard through a **speaker system** or a **headphone**. **Microfilm** technology can be used to directly get miniature outputs from computers on films. Due to large data compression ratio, huge volumes of data can be stored on microfilms and can be viewed using a projector. **CAD-CAM** (Computer Aided Design and Computer Aided Manufacturing) technology can directly convert the drawing instructions in a computer to a set of electrical signals which can be used to drive mechanisms to automatically manufacture products.

2.5 Memory Unit

To process data, both the data to process and the instructions to process them need to be stored in the computer. All **Computer data and instructions can be stored** in special locations of the computer called the computer memory. Depending upon the type of storage, two basic types of memories are **Primary Memory** and **Secondary Memory**.



Primary Memory

1. **Primary Memory**: The primary memory is the **main memory of the computer** and a computer cannot run without it. Two types of primary memory are Random Access Memory (**RAM**) and Read Only Memory (**ROM**). **RAM is a volatile memory** that is used to temporarily store instructions and data that are needed during a program execution. It is the working memory of the CPU and stores temporary data, but the contents are erased if the power is switched off. Whereas **ROM is a non-volatile memory** that stores instructions that are required by the computer during start-up. The computer reads the instructions permanently stored in the ROM every time power is on.



Secondary Memory

2. **Secondary Memory**: However to **store large volumes of data permanently** we need a secondary storage device. The secondary memory is a **permanent memory**, where the results of data processing are stored from the RAM for future use. Unlike a RAM, one can store both programs and data permanently in a secondary storage. In case permanent storage of data is not required, **secondary memory becomes optional**. Therefore a computer may run without permanent storage, but it cannot work without primary memory. Magnetic storage media like hard disk, optical media like CD's, DVD's and semiconductor flash memories are examples of secondary storage.

Primary Memory forms the main memory of a computer and a computer cannot run without it. It consists of the RAM and ROM

Secondary Memory is used to store large volumes of data permanently in a computer

Difference between Primary and Secondary Memory

Primary Memory	Secondary Memory
Usually volatile memory	Is a non-volatile memory
Data is lost in case power is switched off	Data is permanently stored even if power is switched off
It is the working memory of the computer	It is used for storage of data in the computer
A computer cannot run without primary memory	A computer may run without secondary memory
Examples are RAM and ROM	Examples are hard disk, CD-ROM, DVD, Flash drive

Primary memory of the computer is composed of **memory cells** where each cell can store either a '0' or a '1'. The arrangement of **8 such cells forms a memory byte**. In all modern computers, data is read or written to the memory in groups of several bytes. Such a **group of bytes form a memory word**. A word length can be of 32 bits or 64 bits.

During a **Read** operation, the data stored in the memory cells gets transferred to the data lines. During a **Write** operation, the data is transferred from the data lines to the memory cells. Whereas a **read operation is a non-destructive one** and retains the previous data, the **write operation is a destructive operation** whereby the previous data is lost after a write operation.

Earlier **Magnetic Ferrite Core Memories** were used to design primary memory. During the 1970s **Semiconductor Memory** replaced these and is presently used to construct both RAM and ROM.

• Read Only Memory (ROM)

Read Only Memory (ROM)

ROM or Read Only Memory is a **non-volatile permanent memory**, the contents of which can be only read but cannot be altered. Each computer has the **ROM BIOS**, which permanently stores the basic information needed by the computer during start-up after power is switched on. This is also called Firmware.

ROM is a non-volatile permanent memory used mainly to store the basic firmware in a computer

There are various types of ROMs available. A common feature for all these is their **ability to retain data and programs forever**, even during a power failure. As described below, different types of ROMs are distinguished by the **methods used to write new data** to them and the **number of times they can be rewritten**.

1. **ROM**: These are the first semiconductor ROMs that contained a **pre-programmed set of data or instructions**.
2. **PROM (Programmable ROM)**: It is purchased in an un-programmed state. Special equipment called a device programmer is used to write the data into the PROM. The data is written by **applying an electrical charge to the input pins** of the chip. Once a PROM has been programmed in this way, its contents are permanent.
3. **EPROM (Erasable and Programmable ROM)**: It is programmed in exactly the same manner as a PROM. However unlike a PROM, EPROMs **can be erased and reprogrammed repeatedly**. To erase an EPROM it is exposed to a strong source of ultraviolet light. By doing this the entire **chip is reset** to its initial un-programmed state. (It is also sometimes called an **UVEPROM**).
4. **EEPROM (Electrically Erasable and Programmable ROM)**: Internally, they are similar to EPROMs, but instead of using UV light, the **erase operation is done electrically**. Any byte within an EEPROM can be erased and rewritten without the need to reprogram the whole chip, as in an EPROM. (It is pronounced as E^2 PROM).

Presently ROM memory is sometimes replaced with flash memory that enables upgrading the firmware

Comparison Chart for ROMs

ROM Comparison Chart

Type	Writable	Cost / Byte	Speed
ROM (Read Only Memory)	No	Inexpensive	Fast
PROM (Programmable ROM)	Only once	Moderate	Fast
EPROM (Erasable and Programmable ROM)	Many times	Moderate	Fast
EEPROM (Electrically Erasable and Programmable ROM)	Many times	Expensive	Fast to read, slow to erase/write

• Random Access Memory (RAM)

Random Access Memory (RAM)

Programs are loaded into the main memory and run from there. Data is also loaded into the memory for faster access. The type of memory in which all these operations are done is known as the Random Access Memory or **RAM**. It is so called because the amount of time required to access data stored in the RAM is the same, independent of the location of the data in the RAM. That is, the **data can be randomly accessed** with the same access time. Whereas a write operation on a RAM is destructive, a read operation is non-destructive. However a RAM is **volatile in nature** and **requires a regular supply of power to refresh its contents**. In the absence of any refreshing supply or when the computer is switched off, the content of the RAM is lost. These are available as IC packages commonly called **memory sticks** in **512MB**, **1GB**, or **2GB** modules.



RAM is a volatile primary memory used to store data temporarily during the working of a computer

Depending upon its retentive power, a RAM can be broadly classified into a **Static RAM** and **Dynamic RAM** as described in the next page.

DRAM

1. **Dynamic RAM (DRAM):** A Dynamic RAM cell is a **volatile memory** that usually consists of a **transistor and a capacitor** and the **electrical charge needs to be periodically refreshed** to retain its data.

A DRAM cell has a very short data lifetime of about 4 milliseconds. To retain the data for a longer time, the DRAM chip is refreshed at regular intervals of time before the data is lost. **As the memory cell needs to be recharged at regular intervals of time, such a RAM is called a dynamic RAM.** It is possible to fabricate a large number of these cells in a small area, resulting in a **high bit density** and lower cost. Due to their low price and high packing density, these are used for the bulk manufacturing of RAM in the form of IC chips of 512MB, 1GB, 2GB, 4GB capacity.

SRAM

2. **Static RAM (SRAM):** Unlike a DRAM, SRAM is a type of RAM which can **retain its contents** as long as electrical power is applied to the chip. However, like DRAM if power is switched off, its contents will be lost forever. These are made using special memory elements called memory latch. Since these RAM do not require periodic refreshing of the existing data, these are called static RAM and are much **faster than Dynamic RAM**. However these memories have **lower packing densities** and hence are **costly**. Therefore SRAM is used in applications, where faster memories are required (as in cache memories), but are used in smaller quantities like **512KB**, **1MB**, or **2MB**.

Comparison b/w DRAM & SRAM

Comparison between DRAM and SRAM

DRAM	SRAM
It is a volatile memory.	It is also a volatile memory.
Basic memory cell consists of a transistor and capacitor.	Basic memory cell consists of several transistors only.
Needs to be refreshed at regular intervals of time to retain its data.	No need to refresh at regular intervals of time to retain its data.
Packing density is high.	Packing density is low.
Cost per bit is low.	Cost per bit is high.
Speed is of the order of 80 nanoseconds.	Speed is of the order of 20 nanoseconds.
Used to make main working memory of computer	Used to make cache memory of a computer

First commercially available DRAM chip, Intel-1103, was introduced in October 1970

Difference b/w RAM & ROM

Difference between RAM and ROM

RAM	ROM
Full form is Random Access Memory.	Full form is Read Only Memory.
It is a form of temporary memory where the memory content is lost if power is switched off.	It is a form of permanent or non-volatile memory where the contents are stored permanently.
It is usually used to form the working memory of the computer.	It is usually used to form the BIOS chip of the computer.
One can both read and write data to a RAM	One can only read data from a ROM

Software can partition a portion of a computer's RAM to allow it to work as a much faster hard-drive that is called a RAM disk

Apart from the computing power, the other most important thing that has added to the speed and power of a computer is its RAM content. During the 70's, a RAM content of 64KB was considered high. However with development in multimedia and game technology, the requirement for RAM has gone up. Compared to 64KB, a present day general computer (as of 2014) has 2 GB to 4 GB of RAM. That means an **increase of 1 million times** in RAM capacity over the past 30 years! All this has been possible with the introduction of newer and better RAM types over the years.

Various types of RAM packages that have been developed over the years include **DRAM** (Dynamic RAM), **EDO RAM** (Extended Data Out RAM), **SRAM** (Static RAM), **SDRAM** (Synchronous DRAM), **RDRAM** (Rambus DRAM), **DDR SDRAM** (Double Data Rate SDRAM), **DDR2 SDRAM** (Double Data Rate 2 SDRAM), etc.

Cache Memory

- **Cache Memory**

Cache memory is also a **type of volatile memory like RAM but is much faster** than RAM. It is basically made **using static RAM technology** and hence the access time of cache memory is much faster than that of the RAM. It is thus used to temporarily store active data and instructions during processing and **functions in between the CPU and the main memory** i.e. the RAM.

As most programs contain a number of iterative loops (such as while and for), if an instruction is executed there is a likelihood of the same instruction being executed again soon. Similarly while working with tables, records or arrays of data, the processor works with the same set of data over a short period of time. This type

of **instruction and data that is needed frequently is stored in the cache memory** of a computer for faster access.

When a program is running, and the CPU wants to read an instruction or data, it first checks if the same is available in the cache. If it is found, it is delivered to the CPU. Otherwise a **block of memory including the requested information is fetched** from the main memory. This block of instructions is then copied to the cache memory and delivered to the CPU. During execution of the program it is likely that there will soon be references to other words in the block recently stored in the cache. Thus the next time the CPU attempts to read a word it is very likely that it will find the same in the cache. This saves the time to again access the data from the main memory.

A computer system can **contain more than one cache memory**. For example the Pentium group of processors have two different caches referred to as the **L1 and L2 cache**. The L1 cache is embedded into the processor chip while the L2 cache is located outside the processor chip.

Cache Memory is a type of volatile memory, but much faster than RAM. It is basically made using static RAM (SRAM).

Buffer Memory

Buffer Memory

The term buffer memory in a computer is used to refer to an area of the **memory that is used for temporary storage of data**, while it is being moved from one place to another. Buffers are typically used when there is a **difference between** the **rate at which data is received** and the **rate at which it can be processed**. Usually the data is temporarily stored in a buffer as it is retrieved from an input device (such as a keyboard) or just before it is sent to an output device (such as printer/speakers).

Buffer Memory in a computer is used for temporary storage of data, while it is being moved from one place to another

Buffers can be created in a fixed memory location in hardware. However, a **majority of buffers are implemented in software**, which typically uses the faster RAM to store temporary data. This is helpful as the access time of RAM is much faster compared to hard disk drives.

Among the various applications of buffers, these are **used by printers to store the processed data** from the computer. It is then supplied to the printer as per the speed of the printer. Another example of buffer **use is in streaming online multimedia content**. Without the use of a buffer, one would either have to wait for the entire file to load to the local machine before playing it, or have playbacks with breaks and jumps created by millisecond (or longer) delays in the incoming data stream.

• Memory Registers

Memory Registers

Apart from cache memory, the processor contains several special **high speed memory registers located inside the Control Unit and the ALU**. These are used to store the current instruction and data, address of the next instruction to execute and the immediate result of any calculation done by the ALU. A register can be 8, 16, 32 or 64 bits in size. This is sometimes called the **word size**. The **processor directly communicates with these memory registers** to access data and instructions and hence, more is the word size, faster the data can be processed by the CPU.

Names and functions of some important registers are given below:

Names of some special registers

- **Memory Address Register (MAR)**: Used to hold the address of the current memory location
- **Instruction Register (IR)**: Holds the current instruction that is being executed
- **Program Counter (PC)**: Holds the address of the next instruction to be executed
- **Input / Output Registers:** These are used to communicate with the input/output devices
- **Accumulator (ACC)**: It is used to store the result of any arithmetic operation in the ALU
- **Memory Buffer Register (MBR)**: This is a buffer register used by the CPU to store data which is being transferred to or from an immediate storage (like RAM).

Data can enter or leave a register either serially or in parallel. Thus there are four categories of registers depending upon the way the **data enters and leaves a register**. These are Serial-In-Serial-Out (**SISO**), Serial-In-Parallel-Out (**SIPO**), Parallel-In-Serial-Out (**PISO**), and Parallel-In-Parallel-Out (**PIPO**).

2.6 The Processing Unit

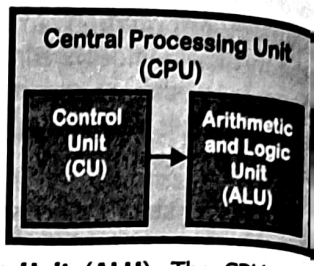
The Processing Unit

The **processor** is the brain of the computer and is involved in **doing all the calculations that take place whenever any data is processed**. To execute a program, the processor should have the following properties as described in the next page.

- It should **read and write information** (program and data) from and to the memory.
- It should recognise and **perform a series of simple commands** to carry out the programs.
- It should **co-ordinate between different parts** of the computer to carry out instructions

CPU is the brain of the computer and consists of the ALU and the CU

For personal computers the **main processor is known as a microprocessor** (or μp) which is a single chip mounted on the motherboard. For larger computers, a single microprocessor may be replaced by several parallel processors which work together and hence increases the total processing power of the computer. These are collectively known as the **Central Processing Unit** or **CPU**.



The CPU is the brain of any computer system and is made up of two basic components, namely the **Control Unit** (**CU**) and the **Arithmetic and Logic Unit** (**ALU**). The CPU also contains a set of **memory registers** to temporarily store instruction and data, during processing.

Each processor has its own **instruction set** built into its hardware. It uses these instructions to process data. The instructions to carry out to process data are given to the computer in the form of a program. **The execution of each instruction is done in three steps**. These are the **Fetch**, **Decode**, and **Execute** operations. The 'fetch' operation is used to load the instruction and data into the main memory of the computer. The 'decode' operation is then used to decode the instruction. The 'execute' operation is finally used to follow the decoded instruction and process the data accordingly. The processed data is then stored back in memory, to be used as required.

Control Unit (CU)

- <u>The Control Unit (CU)</u>

During data processing, the Control Unit carries out the following functions:

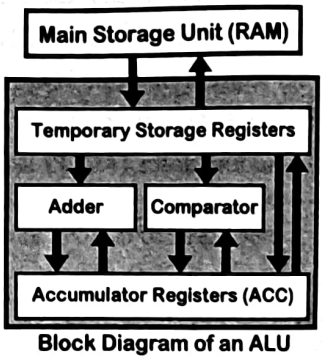
The Control Unit (CU) mainly works as a supervisory unit and issues control signals

- **Control supervision**: The Control Unit acts like a supervisory unit in the CPU and is responsible for issuing control instructions to carry out different functions of the CPU.
- **Activate memory locations**: To read the data and instruction from the main memory, the CU issues control signals to activate those memory locations. Data and instructions are then transferred from the main memory to special purpose registers in the Control Unit.
- **Decode instructions**: The decoder present inside the CU is used to decode an instruction and send appropriate signals to the ALU to process the data.
- **Generate write signal**: The processed data needs to be transferred from the temporary memory registers in the CU to the main memory. The control unit issues a write signal to write the processed data to different memory locations in the main memory.
- **Control Coordination**: The control unit is responsible for coordination between the memory and different input/output devices. As an example, the control unit may load two numbers into the registers in the ALU and tell the ALU to add the two numbers or to check if the numbers are equal.

ALU

- <u>The Arithmetic and Logic Unit (ALU)</u>

The Arithmetic and Logic Unit (ALU) is the place where the **actual execution of the instructions takes place** during data processing. It is responsible for carrying out arithmetic operations (like addition, subtraction, etc.) and different logical operations (like checking equality, >, <, AND, OR, etc.). To carry out these arithmetic and logical operations, the ALU contains some special purpose **registers**, an **adder** and a **comparator** circuit.



Block Diagram of an ALU

The Arithmetic and Logic Unit (ALU) is the place where the actual calculations take place

During a calculation, data is first moved from the main memory (RAM) to temporary storage registers in the ALU. The data present in the storage register and in the Accumulator are then transferred either to the **Adder circuit** or to the **Comparator circuit** for necessary **calculations or comparison**. After an operation the result is stored back in the Accumulator. Finally the result is moved from the Accumulator to the main memory through the storage registers.

The Arithmetic and Logic Unit therefore carries out the following functions:

- **Fetch data**: The ALU fetches the data that needs to be processed and stores them into temporary memory registers for processing.

- **Carry out calculations**: Based on the control signals generated by the Control Unit, the ALU carries out various operations like addition, subtraction, multiplication or division etc.
- **Compare data**: Other than doing calculations, the ALU can also compare two sets of data like check for equality or in-equality based on the control signals generated by the Control Unit.
- **Logical operation**: Logical operations like logical AND, OR, NOT etc. are carried out by the ALU.
- **Store processed data**: The processed data is transferred from the Accumulator to the proper storage locations as per the control signals generated by the control unit.

• Microprocessor

Microprocessor

When the **functions of a CPU are integrated on a single integrated circuit** (IC) using ultra large scale integration (ULSI) technology, then it is called a microprocessor. In a microprocessor trillions of transistors that are used to form the **different components of a CPU are packed into a single chip**. By packing all the parts of the CPU within a single chip increases the processing power of the processor. The first 4-bit microprocessors were used for making electronic calculators during the early 1970s.

Intel's 4004, the first microprocessor, was designed for a calculator

8-bit microprocessors were the first to be developed. The bit number indicates the number of bits the processor can process at a time. Thus an 8-bit microprocessor can process 8 bits at a time. The **world's first 8-bit microprocessor was Intel's 4004 microprocessor** introduced in 1971. It was followed by the 8008 microprocessor in 1972.



Introduced in early 1973, National Semiconductor's **IMP-16 was the first 16-bit microprocessor**. This was followed by **Intel's 8088**, which was used in the first IBM PC. It was the first member of the Intel 80-x86 family (80186, 80286, 80386, 80486) that was used in most modern PCs.

When the functions of a CPU are integrated on a single integrated circuit (IC), then it is called a Microprocessor

The world's first **32 bit microprocessor, the BELLMAC-32A**, was developed at the AT&T Bell Labs in 1982. Intel introduced its 32 bit processor the 80386 during 1985. It became the predominant processor architecture for desktop and laptop computers at that time. AMD's Athlon and Intel's Pentium 4 are best examples of other 32 bit processors.

AMD's **AMD64 microprocessor** introduced in 2003 and Intel's 64-bit architecture the Intel 64 led to the beginning of **64-bit computing**. Both versions can run both 32-bit applications as well as new 64-bit software without any performance penalty.



Multiple core processors were the next development in processor technology. For example a dual core processor is basically two microprocessors built into one IC package. This effectively multiplies the performance of the processor as compared to a single core processor. AMD's Athlon X2, Intel's Pentium D, Centrino Duo, Xeon and Core 2 Duo, Core i3, i5, i7 processors are examples of multi-core processors.

2.7 Storage Devices

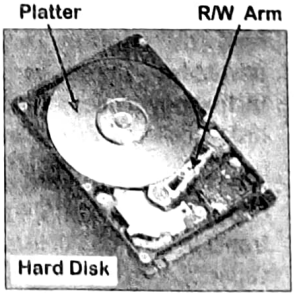
Storage Devices

Since RAM is a volatile type of memory and nothing can be permanently stored in it, to save our data files permanently, we require **secondary memory devices**. In these storage devices the **cost per byte of storage is much less** compared to primary memory. For example, whereas a 2 GB RAM costs around Rs. 1200/-, a 2 TB hard disk drive costs around Rs. 6000/- (as on 2014). Various types of common storage devices are discussed below.

• Magnetic Hard Disks

The first hard drive was created in 1979, and could hold just 5MB of data

Early hard drives with 20MB capacity cost about Rs.50,000/-, whereas a pen drive today holds 8GB memory and costs less than Rs.500/-



Magnetic hard disks are the most common type of **direct access storage devices (DASD)** used to **store data digitally**. In a hard disk, data is recorded permanently on **one or multiple circular disks called platters**. The disk is made of a non-magnetic material like an aluminium alloy over which a thin film of a ferromagnetic material is applied. **Data is recorded by magnetising the ferromagnetic material**.

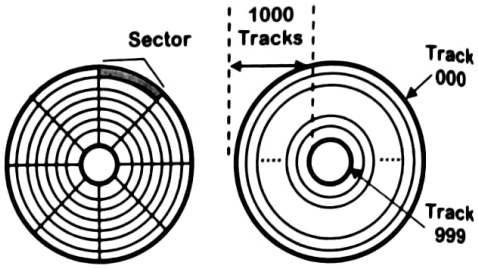
To read or write data onto the disk, a **motor spins the disk at a very high speed** of around 7200 rounds per minute. **A read-write head mounted on an arm** is used to write data onto the disk or read data from the disk. A motor positions the arm containing the read/write head on the disk from where the data is to be read or written.

During writing data, the read-write **head lays down a magnetic pattern representing binary '0's and '1's** on the magnetic disk. When reading from the disk, the read-write head senses the magnetic pattern on the disk and converts them back to binary '0's and '1's.

Magnetic Hard Disks store data permanently on one or more circular disks called platters, coated with a magnetic material

Magnetic hard disks can be used in a computer system only after they are prepared by a process called **disk formatting**. During formatting, a pattern is laid out on the disk which divides the surface of each disk into a number of **invisible concentric logical circles called tracks**. There can be from a few hundred to a few thousand tracks on a single disk surface. The tracks are numbered consecutively from the outermost to the innermost track, starting from 000. In a multiple platter disk pack, the tracks with the same diameter and located on the different surfaces of different platters form a **cylinder**. Thus the total number of cylinders in a disk pack is same as the number of tracks on a single side of a disk. Each track is again further subdivided into **smaller sections called sectors**. Disk drives are designed to read/write whole sectors at a time. Typically there are **4-32 sectors per track** and **512-4096 bytes** per sector. Some operating systems like Windows combine several sectors to form **clusters**.

There can be two types of hard drives based on the type of read/write heads provided. In a **fixed head type disk**, there are individual heads provided for each track of the disk. While in a **moving head type disk** there is only one read/write head provided for each surface of the disk.



Several factors determine the time required to access the data during a read/write operation. The **disk access time** is the **interval** between the time a computer makes a **request** for transfer of data from the disk to the primary memory, and the time this **operation is completed**. To locate the data on the disk, the disk address of the particular data block needs to be given. As the disk address is given by its surface number, track number, and the sector number, the read/write head needs to be positioned based on the above parameters. Based on these, the **disk access time depends upon**:

Seek Time

- **Seek Time**: The time required to **position the read/write head** from the current cylinder or track to the **desired cylinder or track** is called the Seek Time. The seek time varies, depending upon the position of the read/write head when a data request is made. The average seek time is of the order of **10-20 millisecond** depending upon the make or model.

Latency

- **Latency**: Once the cylinder/track is selected and the read/write head is activated, the time required to **position the desired sector under the read/write head** is called **Latency** or **Rotational Delay**. It varies depending upon the initial position of the sector with respect to the read/write head and also upon the rotational speed of the disk, which may be of the order of 7200 rpm. Average latency time is given by the time required for the disk to make half a revolution and is of the order of **5-80 milliseconds**.

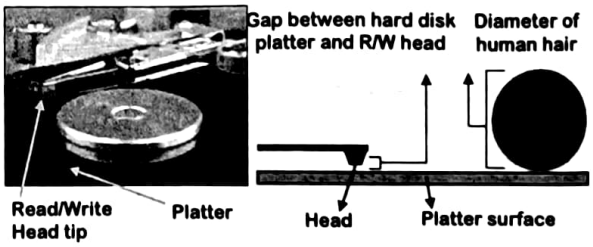
The time required to position the read/write head of a hard disk from the current track to the required track is called Seek Time.

- **Data Transfer Rate**: Once the read/write head is positioned, the **rate** at which the **data is read or written** onto the disk is called the Transfer Rate or Block Transfer Time. It is usually expressed in thousands of bytes per second. The transfer rate depends on the rotational speed of the disk. More the speed more data will get transferred per unit time.

Thus the **Access Time** is the **sum of the Seek Time, Latency, and Data Transfer Time**. The access time needed to locate and transfer a disk block is of the order of 12-60 milliseconds. Of these, the seek time and latency are much higher than the block transfer rate.

The time required to position the required sector under the read/write head is called Latency or Rotational Delay.

The **gap between the head and the platter** is very small and of the order of **several nano-metres** (10^{-9} m) only. Due to this extremely close spacing, even a small dust particle can lead to a hard disk crash. To avoid this, the disks are nowadays stored in an airtight box using a technology called Winchester technology and the disk is called a **Winchester disk**.



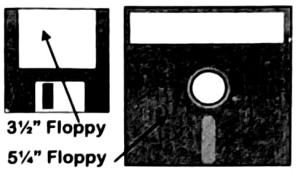
A typical modern hard disk can store between **500 to 2 TB** of data and rotate at 5,400 to 10,000 rpm with an average data transfer rate of 70MB/sec or higher. As of 2014, the highest capacity hard disk available is 2 TB. **Removable external hard disks** are also available for backup.

1 Kilobyte (KB)	= 2^{10} bytes	= 1024 bytes
1 Megabyte (MB)	= 2^{10} KB	= 1024 KB
1 Gigabyte (GB)	= 2^{10} MB	= 1024 MB
1 Terabyte (TB)	= 2^{10} GB	= 1024 GB

• Magnetic Floppy Disks

Floppy Disk

Another type of **direct access storage device** is the **floppy disk**, which is a removable plastic diskette and has a much lower storage capacity. Two different standards of floppy disks were available. The first one was the 5¼" variety with a storage capacity of 1.2MB, and the second was the 3½" one with a storage capacity of 1.44MB. Nowadays **floppy disks have become obsolete**, being replaced by higher capacity flash drives.



Instead of a metallic platter as in a hard disk, floppy disks were constructed using a thin plastic disk coated with a magnetic material. These were available in various capacities depending upon the density of the magnetic material applied on the disk and the number of surfaces used. Accordingly there were **SSSD** or Single Sided Single Density, **SSDD** or Single Sided Double Density, **DSSD** or Double Sided Single Density, **DSDD** or Double Sided Double Density type disks (which have 1.44 MB capacities).

Disadvantages of Magnetic Disk Storage

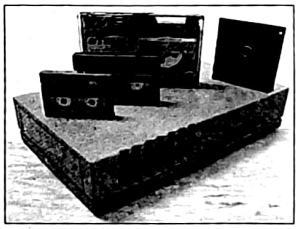
Following are the disadvantages of using magnetic disk storage:

1. **Lower Efficiency for Sequential Processing:** For applications requiring sequential processing of data a sequential device like a magnetic tape is more efficient than a magnetic disk.
2. **Low Security:** It is difficult to maintain the security of information stored in magnetic disks used as a shared secondary storage device.
3. **Prone to Easy Data Loss:** A disk failure or disk crash can result in the loss of the entire information stored in the disk. It is also not easy to recover the lost data.
4. **Higher Cost per Bit than Tape:** The cost per bit of storing in a magnetic tape is lower than a magnetic disk.
5. **Less Portable:** Disk packs like magnetic hard disks are not so easily portable like magnetic tapes though portable hard disks are available at present.
6. **Needs Special Care:** Jerking should be avoided while working on a magnetic hard disk.

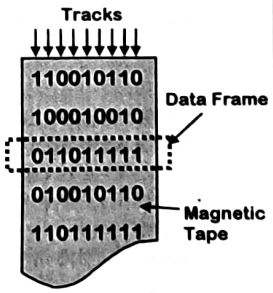
• Magnetic Tape Media

Magnetic Tape

Magnetic tapes have been used for computer data storage for a long time since 1951. Unlike magnetic hard disks, a magnetic tape is a **sequential access storage device (SASD)**. A hard disk drive can move its read/write heads to any random part of the disk platters in a very short amount of time, but to access the nth block in a magnetic tape a tape drive must wind the tape to first skip the first (n-1) blocks to get to the nth block. The sequential access **makes the tape slow** and is not used to store online data. It can be used to store very large files. It is usually **used to keep a backup of the data** from the hard disk, in case of a hard disk crash. A **DAT** (Digital Audio Tape, with capacities of up to 1 TB are available) is basically used for this purpose nowadays.



In a magnetic tape, data is stored as binary digits on a ribbon made of a **polymer material and coated with a magnetic substance**. The width of a magnetic tape varies from ½" – ¼" and is 50 to 2400 feet in length. Data is read or written in parallel tracks along the length of the tape using read/write heads, which change the magnetic property of the tape to store binary '0's and '1's. The tape speed and the data density determines the **data transfer rate** of the tape. The data density is in **bytes per inch** (BPI) or **characters per inch** (CPI) and varies from 1600 to 6250 CPI. The tape speed is expressed in **inches per second** (IPS). Data is usually grouped into blocks separated by a space called **inter-block gap (IBG)**.



Advantages of Magnetic Disk over Tape

Advantages of a magnetic disk over magnetic tape:

a) **Random Access:** Unlike magnetic tapes, magnetic disks support random access of data.

b) **Fast Data Read/Write:** Due to their random access nature, data from magnetic disks can be accessed in few milliseconds which is not possible in case of a sequential tape.

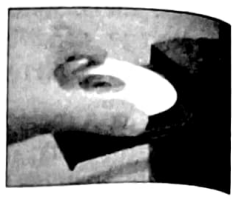
c) **Shared Device:** Due to their random access properties, multiple users often use magnetic disks simultaneously as a shared device. A magnetic tape is not suitable for such purpose.

d) **High Data Transfer Rate**: Data transfer rate of a magnetic disk is usually higher than that of a magnetic tape.

e) **More Durable**: A magnetic tape can snap easily causing unrecoverable loss of data. Similar problems do not arise in case of a magnetic disk.

Compact Disk (CD)

- Compact Disk (CD)

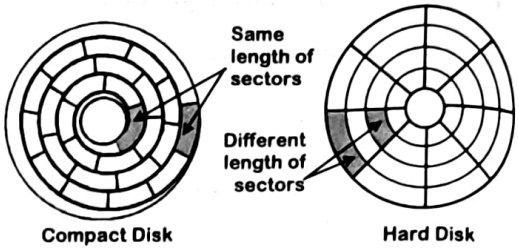


A CD or Compact Disk was originally used to store and playback music in a digital format. Later during the 1980s Sony and Philips developed a standard **to store any type of binary data** in a CD format. Presently a CD is used to store and distribute music, video, data, software etc.

A CD is an optical secondary storage device. It has a capacity of 700 MB

A **CD is a DASD** made of **polycarbonate plastic disk of 12 cm diameter** and 1.2 mm thickness. Smaller 8 cm diameter CDs are also available. The total capacity of a 12 cm CD Read Only Memory or CD-ROM is 700 MB and its audio running time is about 84 minutes.

Data is stored in a CD spirally from the centre of the disk to the circumference, **in sectors**. However, unlike a hard disk sector, the sectors of a CD are of equal length (see diagram). This ensures a higher data storing capacity as there is no wastage of space.

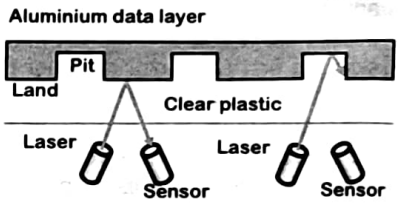


Because the outside area of a spinning disc is moving faster relative to the inside area, the **motor must vary the speed of the CD** to ensure that the surface of the disc is always moving at the same speed underneath the laser read head. This keeps the **linear speed of the track same**.

A CD-ROM drive is used to **read data from a CD**. The drive uses a tray system that slides out to load the disc. To read or write the data a variable-speed motor is used to spin the disk. A CD-ROM read head is a combination of a laser lens, which **focuses a laser beam towards the surface** of the disc and a sensor which determines whether the beam was reflected from the surface of the disk or not. The binary data stored in the CD is read by the **sensor which detects the reflection or absorption of the laser beam** to determine a binary '0' or '1'.

Types of CD

Based on the type of use and the manner in which the '0's and '1's are encoded in a CD, there can be 3 different types of CDs as described below.



1. **CD-ROM**: This type of CD **is a read only media** made of a thin aluminium layer placed between two clear plastic layers. Data is stored on the disc as a **series of land and pit** etched into the clear plastic layer. A land represents a binary 0 and a pit represents a binary 1. A laser light is shown on the reflective aluminium surface of the disc to read the land and pits. When there is a land, the laser gets reflected from the land and is sensed by a sensor. When there is a pit, the laser falls into the pit and gets scattered. The scattered light is not sensed by the sensor (see diagram). This **pattern of changing intensity** of the reflected beam is converted to binary data.
2. **CD-R**: It is a **CD-R**ecordable media, also called a Write Once Read Many (**WORM**) type media, where the user **can write data on the CD only once**, but read it many times. The reflective aluminium layer in such a disk is covered with a special **reactive dye**. To write data, a high power laser is shown on the dye layer. The dye gets discoloured wherever the laser falls. The coloured part can reflect light and the discoloured part scatters light. Hence the coloured and discoloured surface of the dye can be used to represent the land and pit of a normal CD-ROM. The CD is then read by a low power laser.
3. **CD-R/W**: This type is a **CD R**e-**W**ritable media, which can be both written and read many times. The reflective aluminium layer in such a disk is covered with an **amorphous crystalline material**. Initially the crystalline material is transparent in nature. To write data, a high power laser is used to **change the transparent crystal to opaque**, where the light falls. The transparent and opaque surface is then used to represent the land and pit of a CD-ROM. To re-write data, the CD-RW has to be formatted to the clear crystal state again, before new data can be written. The CD is read by a low power laser light.

• Digital Versatile Disk (DVD)

DVD

A DVD or a Digital Versatile Disk (also called a Digital Video Disc) is an **optical storage media** like a CD, but **stores much more data than a CD**. The maximum capacity of a dual layer DVD is about 8.54 GB as compared to a CD of 700 MB capacity (more than 10 times!). Its main uses are **video and data storage**. DVDs are of the same dimensions as compact discs and are available as the standard 12 cm type and the 8 cm mini-DVD type mainly used for video cameras.



A **650 nanometre wavelength laser** light (unlike the 780 nm laser used in a CD) is used for reading data from the DVD. The lower wavelength allows for a **higher data density** in a DVD and an increased storage capacity. Data is written on a DVD using the same technique of land and pits used in a CD.

A DVD is an optical secondary storage device. It is available with a capacity of 4.7 GB and 8.54 GB

DVDs are available as single layer with **4.7 GB** capacity and as dual layer with **8.54 GB** capacity. The **dual layer** version **uses a second physical layer** within the disc to store the extra data. It is made of a thick polycarbonate disk providing the foundation. This layer is followed by a thin opaque reflecting layer and a thin transparent film. The final layer is a clear protective plastic layer. In a dual layer DVD, **data is stored both in the transparent film layer and the reflecting opaque layer** in opposite spiral paths as shown in the diagram to the right. In such a disk, the lower layer data spiral starts from the centre, and the upper layer data spiral starts from the circumference where the upper layer data spiral ends. The drive with dual layer capability **accesses the second layer by focussing the laser through the first semitransparent layer** onto the opaque layer.



• Blu-ray Disk (BD)

Blu-ray Disk

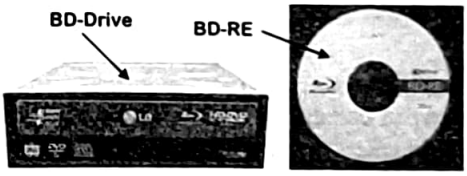
Blu-ray Disc (or BD) is an **optical disc storage medium** that can have a capacity of **up to 50 GB**. The first Blu-ray Disc rewritable drive for the PC was marketed during July 2006. Its **main uses are high-definition video and data storage**. The physical dimension of a BD is same as a standard CD and is 12 cm in diameter.



A BD is an optical secondary storage device. It is available with a capacity of 25 GB and 50 GB

The name Blu-ray Disc is derived from the **blue laser** that is used to read data from the disk. The shorter wavelength ensures that more data can be written in a compact form than a CD or DVD within the same space. Blu-ray Discs also use a better data encoding method that further increase the storage capacity. Data is written onto a BD **using the same mechanism of land and pit** as used in a CD/DVD.

Just like a single and dual layer DVD, a BD is also available as a **single layer 25 GB type** and a **dual layer 50 GB type.**. The theoretical maximum speed for Blu-ray Discs is about 12x. At this speed, data can be **read at 54MB/sec**. Further high speeds in excess of 10,000 rpm cause the disk to wobble and the reading becomes unstable.



Other than a BD-ROM, **recordable** (BD-R) and **rewritable** (BD-RE) blu-ray disks are also available. BD-R discs can be written to only once, whereas **BD-RE can be erased and re-recorded multiple times**.

• Flash Memory

Flash Memory

Flash memory is a **solid state storage device**, which means that there are no mechanical moving parts and everything is done electronically. It **works similar to an EEPROM** and uses a **special transistor** which makes the flash memory non-volatile in nature, that is, the data is not lost even if the power is switched off. The read time of a flash memory is about 10's of nano seconds per byte and writing time is about several microseconds. Typical sector sizes in a flash device are in the range 256 bytes to 16KB. It has also a **high storage density** and is available at a **low cost**. However unlike an EEPROM, flash devices can only be erased one sector at a time and not byte-by-byte.

Flash Memory is a solid state storage device, which means that there are no mechanical moving parts and everything is done electronically

Flash memory is nowadays **used in various applications like in the computer's BIOS chip**, SmartMedia, CompactFlash (which are usually found in digital cameras), Pen Drives (used as a removable storage device for general data backup and storage), memory cards used as solid-state memory disks in laptops. Some of these are discussed in the next page.

1. **SmartMedia**: SmartMedia format developed by Toshiba was launched in 1995 as a **successor to the computer floppy disk** and was called the solid-state floppy-disk card (**SSFDC**). These use flash memory and are available in capacities ranging from **0.5 MB to 128 MB**. The card was the smallest and thinnest of the early memory cards, with less than 1mm in thickness and was popular in **digital camera storage**.

2. **Compact Flash**: These cards were developed by Sandisk in 1994. The largest Compact Flash cards commonly available currently are the **32 GB models** from various manufacturers. These cards are 43 mm wide and 36 mm long and have **transfer rates of 45MB/sec**. (read) and 40MB/sec. (write). Unlike a SmartMedia, these use a controller chip, which enhances the performance. These are used in digital cameras to **store high resolution digital images**.



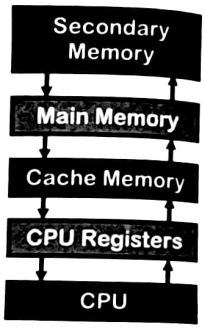
3. **Pen Drive**: A USB flash drive consists of **flash memory integrated with a USB interface**. These are now used as a replacement to the much smaller capacity floppy disks, and are commonly available in 4, 8, 16, 32, 64 GB sizes. Some drives allow 1 million write or erase cycles and have 10-year data retention. Flash drives are also not prone to dust and scratches and are mechanically very sturdy. Hence these are suitable for **transporting data from place to place.** Compared to hard drives, flash drives also use little power, have no moving parts, and are small and light.



Data Access

- Data access from various memory devices

When the CPU wants to access some data, it first looks for it in the **memory registers**. If not found there, it then looks for the data in the **cache memory**. If the data is absent in the cache memory, the CPU next searches for the data from the **RAM**. If the data is not present in the RAM, it has to be loaded from the secondary storage devices like the **Hard Disk**. Back-up media like CD, DVD, etc. are used to preserve data for future use.

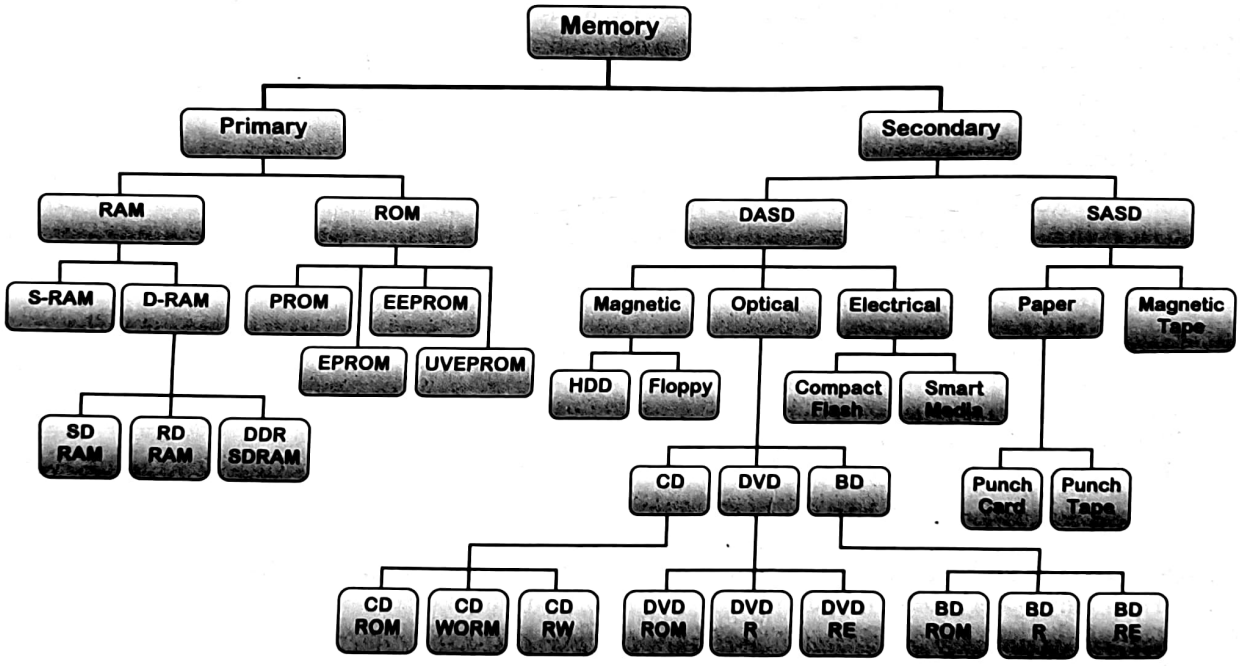


The storage **cost per bit of backup devices is the least**. As the cost-per-bit increases, the memory size also decreases. Thus whereas secondary storage like HDD and tape drives are available in capacities of even up to 2 TB, primary memory like RAM is used in the range of 2 to 4 GB, whereas cache memory is available in the range of 2 to 4 MB only (as on 2014).

The following chart gives a **classification of the various types of primary and secondary memory** in use. Some of these, like punched card and punched tapes are obsolete now, whereas others like **Blu-ray Disks** are recently available in the market.

Computer Memory Classification

Primary and Secondary Memory Classification



2.8 Communication Bus

The **System Bus** serves as the main highway that **connects the different parts of a computer**. These are a set of parallel wires through which data, address and control information flows from one part of the computer to another. The **system bus** can be broadly classified into the **internal bus** and the **external bus**. The internal bus is used to connect the internal components of the CPU i.e. the ALU with the set of registers and cache memory. The external bus on the other hand connects the CPU with the external memory and the input/output systems.

A Bus is a set of electrical paths which carry different types of signals in a computer system.

There are **various devices that need to be connected in a computer system**. One can connect these devices to the CPU using different communication buses. However, using a **common bus** it is **easier and cheaper** to make new connections to the same set of lines than using separate lines for separate devices. Since only a particular subsystem can use the bus at any time, proper rules need to be followed to allow a particular subsystem to use the common communication bus at a time.

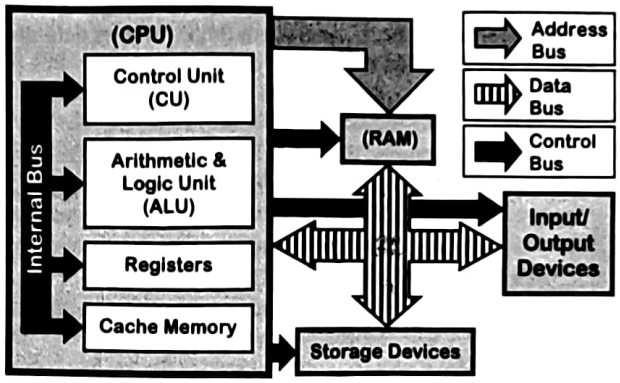
The bus is implemented on the motherboard as a set of parallel wires etched onto the motherboard. On a processor board, the **bus consists of three main components** namely the Address Bus, Data Bus, and the Control Bus.

1. **Address Bus**: The Address Bus **connects the CPU and RAM and carries the memory addresses**. Every storage location in the memory has a unique address. In order to retrieve some data from memory, it is necessary **to specify the address of the memory location** where the data is stored. The Address Bus is used to carry the address of a memory location whenever data needs to be transferred to or from the memory. The number of wires in the bus determines the number of memory locations that can be addressed by the CPU.

Address Bus

2. **Data Bus**: The Data Bus is the electrical path that **connects the CPU, memory and other hardware locations from where data needs to be transferred**. The number of wires in the bus determines the amount of data that can flow at a given time. Thus a 32 bit bus can transfer four bytes of data, whereas a 64 bit bus can transfer 8 bytes of data at a time.

Data Bus



3. **Control Bus**: In addition to sending addresses and exchanging data with the memory, the CPU also needs **to send control signals** to the memory to specify whether the data is to be read from or written to the specified address location. The control bus thus carries signals, which are in the form of *READ/WRITE* commands. It also sends signals to *ENABLE* particular devices.

Control Bus

On a processor board, the bus consists of three main components namely the Address Bus, Data Bus, and the Control Bus

A memory access by the CPU involves all the three buses – the control bus carries the memory access command, the address bus carries the memory address from where the data needs to be fetched or written, and the data bus carries the actual data to and from the memory.

There are **various bus architectures** like ISA, EISA, PCI, and USB that have evolved over the years. This development was a result of the development of computer hardware and the requirement for higher and higher data transfer rates. The increased data transfer rate was a requirement for the high end graphics and video data that needed to be transferred through these buses in applications like computer games, playing audio and video on computers etc.

- Universal Serial Bus (USB)

USB

This type of bus system was released in 1998 to attach low speed input/output devices to a computer. Using an USB, theoetically upto 127 devices can be attached to a computer. The standard uses the **same type of cable for all types of devices**. An USB cable **consists of 4 wires – two for data, one for power, and one for ground**. Voltage transitions are used to transmit the binary '0's and '1's.

The earliest standard was the version 1.0 that ran at 1.5 Mbps and was suitable for keyboard, mouse, and digital cameras. The latest standard is **version 2.0 which runs at 480 Mbps** and supports printers, digital cameras, USB flash drives, and other high speed peripheral devices. Presently, USB is the **most popular communication bus** used to attach peripheral devices to a computer.

- **Data basically means a collection of facts or raw materials**
- When the raw data is processed as per the requirement of the individual then we get a more useful and meaningful interpretation of the data which is called Information
- **The process of converting data to information is called Data Processing**
- **The data in a system can be the information from another system. The information processed by a system can serve as the data for another system**
- Based on the location of the information, it can be classified into five levels as International Information, National Information, Corporate Information, Departmental Information, and Individual Information
- **The computer stores single pieces of information in units called fields or attributes like ID, name, etc. Several such related fields together form records. Several such records of a similar type form an entity set. A set of records which form an entity set are taken together and stored as a file**
- Data can be broadly classified into Physical Data type and Logical Data type
- **Physical Data indicates the process in which the raw data gets stored in the computer**
- Logical Data indicates the logical arrangement of data as viewed by the user or the application programmer
- **Data processing by a computer involves generally three basic steps viz. Input, Process, and Output and the total process is called a Data Processing Cycle or IPO cycle**
- During the input phase the raw data to be processed is input into the computer system by means of various input devices and stored in various media like the hard disk, RAM etc. before it is processed
- **The process phase is used to actually modify or process the data to get the desired information**
- During the output phase the processed data is returned back to the user. This output can either be displayed to the user, taken in a printed form, or can be stored for further processing
- **In an Electronic Data Processing System (EDPS) computers are used to process data electronically**
- A Management Information System (MIS) involves data processing related to the internal working of a business for proper business management
- **Decision Support System (DSS) is a computerised data processing system that supports a MIS and deals with business decision-making activities**
- Transaction Processing System (TPS) is a system used to process data during a transaction in database system
- **Data processing requires an input device, a processing device, memory, and an output device. These form the basic building blocks of a computer and are collectively called computer hardware**
- Software is a sequence of instructions that needs to be given to the computer in a language that the computer understands to make the hardware work
- **In case a part of the software is permanently stored in a computer in a read only type memory, then such software is called firmware. They keeps information related to the computer system**
- Apart from the computer hardware and software, there is a third entity without which the computer will not run. It is called liveware or humanware and consists of the persons who operate the computers
- **The most common input device is the Keyboard. It resembles a mechanical type-writer where the alphabets are placed in the QWERTY format along with the digits 0 to 9 and punctuation symbols**
- Of the different types of keyboards available, the elastomer-dome-switch keyboards are the most common ones
- **A mouse is a point and click input device used to point and select an item on a Graphical User Interface (GUI)**
- A mechanical mouse and an optical mouse are the two types of mouse available based on the way a mouse determines its position on a surface
- **A scanner is an input device which can be used to input images and documents into the computer**
- Drum scanners are used by the publishing industry and can scan highly detailed images
- **A barcode reader is used to scan ready product information from product labels**
- A touch screen is a display screen where the user can enter data by touching the screen at specific locations
- **A light pen is a point and draw input device which allows the user to point to objects displayed on a CRT. These are also used for computer aided design applications**
- Optical Mark Recognition (also called Optical Mark Reading or OMR) is the process of detecting data from pre-printed document-forms marked by humans
- **Optical Character Recognition (OCR) technology is used to optically scan pages containing text and then to identify and save the scanned data as a text file instead of an image file**
- Magnetic Ink Character Recognition or MICR technology is used mainly by banking systems for faster processing of large volumes of cheques
- **Web cameras or simply web cams are input devices used for video capturing and are connected to a computer through an USB port**
- Card readers are electromechanical devices that were used to read the information punched into a paper card

- **Punched tape readers were electro-mechanical devices that were used to input information punched on a paper tape. These used mechanical sensing pins to read the characters punched in each line of the tape**
- A graphic tablet or simply a tablet is an input device on which the user can draw anything using a special pen and the image is directly transferred to the computer display and can be saved
- **Microphones can be used to directly input audio signals like voice or music into the computer**
- A visual display unit (VDU) is the most widely used output device. It uses a display screen and electronic circuitry to display a picture corresponding to the video signal sent by the computer
- **There are two main technologies used to manufacture a VDU, namely CRT and LCD type displays**
- A colour monitor has three electron guns that correspond to the primary colours red, green and blue and emit streams of electrons that overlap and produce any colour
- **A display adapter is used to generate the video format compatible with a particular monitor**
- In a high resolution colour monitor 8 bits are used per pixel to represent different intensities of the colours red (8bits), green (8bits), and blue (8bits). Thus using 24 bits a total of 224=16.8 million colours can be produced
- **There are two types of LCD's namely Passive Matrix LCD and Active Matrix LCD (or TFT monitor)**
- In an impact type printer, the image is produced on a paper by some electromechanical impact device just like the working of a mechanical typewriter
- **Instead of printing a character at once, a Dot Matrix Printer uses a group of pins of very small radius to strike the ink ribbon and create a dot at the point of impact**
- Daisy Wheel Printer are character printers where a special type of wheel called a Daisy Wheel contains the set of characters used in the printing process
- **Line Printer is an impact printer used to print a complete line of text at a time at a very high speed**
- In a non-impact type printer the image is formed on paper without the need for any hammer or ink ribbon
- **An inkjet printer is a non-impact printer which prints by ejecting microscopic drops of ink through tiny ink nozzles in the print head**
- Laser Printer is a non-impact type printer which uses dry ink (called toner), static electricity and heat to deposit and bond the ink onto the paper
- **A plotter is similar to an inkjet printer, but is used for printing on continuous paper rolls**
- CAD-CAM (Computer Aided Design and Computer Aided Manufacturing) technology can directly convert the drawing instructions in a computer to a set of electrical signals to drive mechanisms to automatically manufacture products
- **The primary memory is the main memory of the computer and a computer cannot run without it**
- Two types of primary memory are Random Access Memory (RAM) and Read Only Memory (ROM)
- **RAM is a volatile memory that is used to temporarily store instructions and data that are needed during a program execution**
- ROM is a non-volatile memory that stores instructions that are required by the computer during booting
- **The secondary memory is a permanent memory, where the results of data processing are stored permanently**
- One can store both programs and data permanently in a secondary storage
- **PROM (Programmable ROM) is a type of ROM which the user can program only once using a device programmer**
- EPROM (Erasable and Programmable ROM) is programmed similar to a PROM. However unlike a PROM, EPROMs can be erased using UV light and reprogrammed multiple times
- **EEPROMs (Electrically Erasable and Programmable ROM) are similar to EPROMs, but instead of using UV light, the erase operation is done electrically**
- A simple Dynamic RAM cell is a volatile memory that usually consists of a transistor and a capacitor and needs continuous refreshing to retain the data
- **SRAM is a type of RAM that retains its contents as long as electrical power is applied to the chip**
- Cache memory is also a type of volatile memory like RAM but is much faster than RAM. It is basically made using static RAM technology and hence its access time is much faster than RAM
- **The term buffer memory in a computer is used to refer to an area of the memory that is used for temporary storage of data, while it is being moved from one place to another**
- The processor contains several special high speed memory registers located inside the Control Unit and the ALU
- **Registers are used to store the current instruction and data, address of the next instruction to execute and the immediate result of any calculation done by the ALU**
- Memory Address Register (MAR) is used to hold the address of the current memory location
- **Instruction Register (IR) is used to hold the current instruction that is being executed**
- Accumulator (ACC) is used to store the result of any arithmetic operation in the ALU
- **The processor is the brain of the computer and is involved in doing all the number processing that takes place whenever any data is processed**

- CPU is made up of the Control Unit (CU) and the Arithmetic and Logic Unit (ALU)
- **The Arithmetic and Logic Unit (ALU) is the place where the actual execution of the instructions take place during data processing**
- Each processor has its own instruction set implemented into its hardware. It uses the instructions available in its instruction set to process the data
- **When the functions of a CPU are integrated on a single integrated circuit (IC) using ultra large scale integration (ULSI) technology, then it is called a microprocessor**
- Magnetic hard disks are the most common type of direct access storage devices used to store data digitally
- **Data is recorded on a hard disk by magnetising the ferromagnetic material covering of the hard disk platter. The direction of magnetisation is used to represent either a '0' or a '1'.**
- When a new file is created, the location of the file is written in a table on a specific location on the hard disk called the File Allocation Table or FAT
- **Magnetic hard disks can be used in a computer system only after they are prepared by a process called disk formatting**
- During formatting, a pattern is laid out on the disk which divides the surface of the disk into a number of invisible concentric logical circles called tracks
- **Each track is further subdivided into smaller sections called sectors**
- Seek Time is the time required to position the read/write head from the current track to the desired track
- **Latency or Rotational Delay is the time required to position the desired sector under the read/write head once the cylinder/track is selected and the read/write head is activated**
- To avoid contamination, the hard disk platters are nowadays stored in a sealed airtight box using a technology called Winchester technology and the disk is called a Winchester disk
- **Another common type of direct access storage device is the floppy disk, which is a removable plastic diskette and has a much lower storage capacity of 1.44 MB. These are obsolete now**
- Unlike magnetic hard disks, a magnetic tape is a Sequential Access Storage Device (SASD)
- **In a magnetic tape data is stored as binary digits on a ribbon made of a polymer material and coated with a magnetic substance**
- Data is stored in a CD spirally from the centre of the disk to the circumference, in sectors. However, unlike a hard disk sector, the sectors of a CD are of equal length
- **A CD-ROM read head is a combination of a laser lens, which focuses a laser beam towards the surface of the disc. A sensor determines whether the beam was reflected from the surface of the disk or not**
- A DVD or a Digital Versatile Disk (also called a Digital Video Disc) is an optical storage media like a CD, but stores much more data. The maximum capacity of a dual layer DVD is about 8.54 GB
- **DVDs are available as single layer with 4.7 GB capacity and as dual layer with 8.54 GB capacity**
- Blu-ray Disc (or BD) is an optical disc storage medium that can have a capacity of up to 50 GB
- **Flash memory is a solid state permanent storage device where data storage is done electronically**
- The System Bus serves as the main highway that connects the different parts of a computer
- **The internal bus connects the internal components of the CPU i.e. the ALU, the set of registers, cache memory**
- The external bus connects the CPU with the external memory and the input/output systems
- **The Address Bus connects the CPU and RAM and carries the memory addresses**
- The Data Bus is the electrical path that connects the CPU, memory and other hardware locations from where data needs to be transferred
- **Control Bus is used to send control signals to the memory to specify whether the data is to be read from or written to the specified address location. It also deals with enabling specific hardware**
- Universal Serial Bus (USB) is used to attach low speed input/output devices to a computer

Review Questions

Q1. **Multiple Choice Questions. Select from any one of the four options.** **1 each**

i) Raw data is processed to get a more meaningful interpretation of the data which is called:
a. Constant b. Information c. Text d. Variable

ii) The process of converting data to information is called:
a. Data Hiding b. Data Encapsulation c. Data Processing d. Data Formatting

iii) A computer stores single pieces of information in units called:
a. Attributes b. Rows c. Tables d. Files

iv) Data processing by a computer involves generally three basic steps viz. Input, Process, and:
a. Output b. Display c. Storage d. None of these

v) EDPS stands for:
a. Electrical Data Processing System b. Electronic Data Processing System
c. Electronic Digital Processing System d. Electronic Data Procuring System

vi) MIS stands for:
a. Magnetic Information Storage b. Management Information Storage
c. Management Integration System d. Management Information System

vii) DSS stands for:
a. Decision Support System b. Design Support System
c. Decision Storage System d. Data Support System

viii) TPS stands for:
a. Token Processing System b. Transaction Processing System
c. Transfer Protocol System d. Topological Processing System

ix) A sequence of instructions that needs to be given to the computer in a language that the computer understands is called a:
a. Software b. Hardware c. Firmware d. Livewire

x) In case a part of the software is permanently stored in a computer in a read only type memory, then such software is called:
a. Hardware b. Software c. Livewire d. Firmware

xi) In a standard computer keyboard the alphabets are placed in:
a. QWERTY order b. PQRST order c. ABCDE order d. MNOPQ order

xii) A mouse is a point and click input device used to point and select an item on a:
a. CLI b. CUI c. GUI d. None of these

xiii) A _____ is an input device which can be used to input images and documents into the computer:
a. MICR b. OCR c. Bar Code Reader d. Scanner

xiv) A _____ is an input device used to scan ready product information from product labels:
a. Scanner b. MICR c. OCR d. Bar Code Reader

xv) OMR stands for:
a. Optical Monitoring Reader b. Opto Mechanical Reader
c. Optical Mark Recognition d. Optical Mark Register

xvi) OCR stands for:
a. Optical Code Recognition b. Optical Character Resolution
c. Optical Colour Recognition d. Optical Character Recognition

xvii) MICR stands for:
a. Magnetic Ink Colour Recognition b. Magnetic Input Character Recognition
c. Magnetic Ink Character Recognition d. Magnetic Ink Code Recognition

xviii) VDU stands for:
a. visual data unit b. visual display unit c. virtual data unit d. virtual display unit

xix) CRT stands for:
a. Cathode Ray Tool b. Colour Ray Tube c. Cathode Ray Tube d. Cathode Ring Tube

xx) LCD stands for:
a. Light Colour Display b. Liquid Crystal Display c. Light Crystal Display d. Logical Circuit Diagram

xxi) In a high resolution colour monitor how many bits are used per pixel colour?
a. 8 b. 16 c. 32 d. 64

xxii) TFT stands for:
a. Tin Film Transistor b. Total Film Transistor c. Tight Film Transistor d. Thin Film Transistor

xxiii) In an impact type printer, the image is produced on a paper by:
a. wet liquid ink b. mechanical impact
c. electromechanical impact d. dry ink powder

xxiv) DMP stands for:
a. Dot Matrix Printer b. Desk Mode Printer c. Dot Mode Printer d. Desk Matrix Printer

xxv) Line Printer is an impact printer used to print:
a. a complete line b. a complete word c. a complete paragraph d. a complete picture

xxvi) In which printer type an image is formed on paper without the need for any hammer or ink ribbon?
a. DMP b. Laser c. Daisy Wheel d. Drum Printer

xxvii) Laser Printer is what type of a printer?
a. non-impact b. impact c. line d. character

xxviii) Two types of primary memory are:
a. RAM and ROM b. RAM and Cache c. ROM and Cache d. All of these

xxix) Secondary memory is a:
a. volatile memory b. non-volatile memory c. static memory d. dynamic memory

xxx) PROM stands for:
a. Programmable Read Only Memory b. Parameterised Read Only Memory
c. Process Read Only Memory d. Producible Read Only Memory

xxxi) EPROM stands for:
a. Electronically Programmable ROM b. Electro-optically Programmable ROM
c. Erasable and Programmable ROM d. Electrically Programmable ROM

xxxii) EEPROM stands for:
a. Electronically Erasable and Programmable ROM
b. Electrically Erasable and Programmable ROM
c. Electro-mechanically Erasable and Programmable ROM
d. Electro-optically Erasable and Programmable ROM

xxxiii) DRAM stands for:
a. Dynamic RAM b. Double RAM c. Dielectric RAM d. Data RAM

xxxiv) The term used to refer to an area of the memory that is used for temporary storage of data:
a. HDD b. ROM c. buffer memory d. CD

xxxv) ACC stands for the register:
a. Accumulator b. Actual register c. Accumulation register d. Alternate register

xxxvi) The full form of ALU is:
a. Arithmetic and Logical Unit b. Accumulation and Logic Unit
c. Arithmetic and Logic Unit d. Arithmetic and Legal Unit

xxxvii) Data is recorded on a hard disk using what physical property of a material?
a. Speaker b. Electrical c. Optical d. Magnetic

xxxviii) Magnetic hard disks can be used only after they are prepared by a process called:
a. disk processing b. disk formatting c. disk manufacturing d. disk editing

xxxix) During formatting, a pattern is laid out on the disk which divides the surface of the disk into a number of invisible concentric logical circles called:
a. sector b. track c. platter d. surface

xl) Each track is further subdivided into smaller sections called:
a. track b. cylinder c. surface d. sector

xli) Time required to position the read/write head from the current track to the desired track is:
a. Access Time b. Seek Time c. Rotational Time d. Transfer Time

xlii) Time required to position the desired sector under the read/write head once the track is selected.
a. Transfer Time b. Seek Time c. Latency d. Access Time

xliii) To avoid contamination, the hard disk platters are nowadays stored in a sealed airtight box using a technology called:
a. Magnetic Disk technology b. Vacuum Disk Technology
c. Hard Disk technology d. Winchester technology

xliv) Unlike magnetic hard disks, a magnetic tape is a:
a. SASD b. DASD c. RASD d. None of these

xlv) CD stands for:
a. Complete Disk b. Compound Disk c. Colour Disk d. Compact Disk

xlvi) DVD stands for:
a. Dynamic Versatile Disk b. Digital Versatile Disk c. Digital Vinyl Disk d. Digital Versatile Data

xlvii) BD stands for:
a. Blue-ray Disc b. Blu-ray Disc c. Blue-record Disc d. Blue-ray Data

xlviii) The electrical path that connects the CPU and RAM and carries the memory addresses:
a. Internal Bus b. Data Bus c. Address Bus d. Control Bus

xlix) The electrical path that connects the CPU, memory and other hardware locations from where data needs to be transferred:
a. Control Bus b. Address Bus c. Data Bus d. Internal Bus

l) USB stands for:
a. Uniform Serial Bus b. Universal Serial Bus c. Universal Signal Bus d. Universal Serial Bay

Q2. **Short Answer type questions:** **1 each**

i) What do you mean by software?
ii) What do you mean by firmware?
iii) What is liveware?
iv) What do you mean by information?
v) State one difference between data and information.
vi) State the components of a data processing cycle.
vii) Name any two input devices other than keyboard and mouse.
viii) Name the two different types of mouse technologies available.
ix) What is the use of a scanner?
x) What is the use of a bar code reader?
xi) Write the full form of OMR.
xii) Write the full form of OCR.
xiii) Write the full form of MICR.
xiv) Write the full form of LCD.
xv) State one advantage of using OCR.
xvi) State one use of OMR.
xvii) What is the use of a web cam?
xviii) Name the two technologies available for video display units.
xix) State one difference between a CRT and an LCD monitor.
xx) Name any two types of impact printers.
xxi) Name any two types of non-impact printers.
xxii) State one difference between an impact and a non-impact printer.
xxiii) State one difference between a printer and a VDU.
xxiv) Define primary memory.
xxv) Name the two types of primary memory.
xxvi) State one difference between RAM and ROM.
xxvii) State the names of any two types of ROMs.
xxviii) Write the full form of DRAM.
xxix) Write the full form of SRAM.
xxx) State one difference between DRAM and SRAM.
xxxi) Write the full form of EEPROM.
xxxii) Name the two components of a CPU.
xxxiii) Write the full form of ALU.
xxxiv) State one function of the ALU.
xxxv) State one function of the CU.

xxxvi) State one function of CPU.
xxxvii) Write the names of any two registers available within the CPU.
xxxviii) State one difference between DASD and SASD.
xxxix) What is a track with respect to a hard disk?
xl) What is a sector with respect to a hard disk?
xli) What is seek time with respect to a hard disk?
xlii) What is rotational delay with respect to a hard disk?
xliii) Write the full form of DVD.
xliv) State the different capacities in which a DVD is available.
xlv) Write the full form of BD.
xlvi) State the different capacities in which a Blu-ray Disk is available.
xlvii) What is Address Bus?
xlviii) What is Data Bus?
xlix) State one difference between an address bus and a data bus.
l) Write the full form of USB.

LA

Q3. **Long Answer type questions:** **7 each**

i) Write short notes on any two types of data processing systems. Draw the block diagram of a computer system. 2+2+3

ii) State 3 differences between data and information. Explain the terms International Information and Departmental Information. 3+2+2

iii) Write a short note on keyboard as an input device. Describe any two types of scanners. 3+2+2

iv) Write a short note on mouse as an input device. Explain the terms OCR and OMR. 3+2+2

v) Write a short note on mouse. Write a short note on light pen. State two uses of OCR. 3+2+2

vi) State any 3 differences between OCR and MICR. State any 4 differences between a VDU and a printer. 3+4

vii) State any 4 differences between an impact and a non-impact printer. State any 3 differences between a printer and a VDU. 4+3

viii) Write short notes on the working of Inkjet printer and Laser printer. What type of printer is a Plotter? 3+3+1

ix) What are character printers? Describe the working of any two types of impact printers. 1+3+3

x) Explain the principle of operation of a CRT monitor. State 4 differences between a CRT and an LCD monitor. 3+4

xi) State the differences between primary and secondary memory. What is cache memory? Name a CPU register. 4+2+1

xii) State any 4 differences between DRAM and SRAM. Write a short note on ROM. 4+3

xiii) State the full names of any 3 different types of ROM available. Briefly explain the terms cache memory and buffer memory. 3+2+2

xiv) State 3 differences between RAM and ROM. State the full names of any two varieties of ROM. State 2 differences between SRAM and DRAM. 3+2+2

xv) State any 3 functions of the Control Unit and any 3 functions of the ALU. What is the full form of ACC register? 3+3+1

xvi) Write short notes on buffer memory and memory registers. Write the full form of MAR. 3+3+1

xvii) Write a short note on a magnetic hard disk used as a storage device. What do you mean by Seek Time and Rotational Delay? 3+2+2

xviii) Explain the terms track, cylinder, sector, cluster, seek time, latency, transfer rate. 1x7=7

xix) Write a short note on the working principle of a CD. How does a DVD differ from a CD? What is a Blu-ray Disk? 3+2+2

xx) What is system bus? Explain the terms Address Bus, Data Bus, and Control Bus. 1+2+2+2

CHAPTER 3
Data Representation

- What is a Number System 3-1
- Conversion of Integer Values from one number system to another 3-2
- Conversion of Fractional Values from one number system to another 3-10
- Arithmetic Operations in various Number Systems 3-16
- Representation of Signed Numbers using Complements 3-24
- Various Binary Coding Schemes 3-32
- Bit map representation of Images 3-34
- Concept of Fixed and Floating Point Numbers 3-35

3.1 What is a Number System?

The basic job of a computer system is to process data. This data can be in the form of numerical value like whole numbers or fractional numbers, graphical, audio, or video data. However all these forms of data are basically stored as numeric data in the computer memory, irrespective of the original data type. When we are dealing with numbers, one of the **basic operations involved is counting**. Let us discuss the various counting methods or number systems that were developed in the course of time.

The Romans used 7 symbols to express their number system. These were I, V, X, L, C, D, and M. There was no symbol for 0.

- Non-Positional Number System



Non-Positional Number System

This is the oldest form of number system used. In this system **various different symbols** are used to represent the numbers. Each symbol used in the number has the same value independent of the position of the symbol in the number. For example 1=**I**, 2=**II**, 3=**III** etc. Each symbol **I** in the number represents the value 1 irrespective of the position of **I** in that number. The **Roman Number System** is an example of this type. The various numbers used in this system are: I, II, III, IV, V, VI, ..., IX, X etc.

The major disadvantage with this system is that **arithmetic operations are very difficult to perform.** This gave rise to the development of the positional number systems.

- Positional Number System



Positional Number System

This type of number system consists of a **fixed set of digits** or symbols that are used to represent the numbers and the **position of a digit** in the number gives the value of that digit. The value of a digit in such a system depends upon the following:

a. The number system used
b. The position of the digit in the number
c. The value of that digit

The base/radix of a number system is the number of different digits present in the number system

The base or radix of a positional number system indicates the number of different digits that are present in the number system to represent the numbers.

Depending upon the **base** or **radix** of a number system, the following **positional number systems** are in use. Each has its own set of digits that it uses to express a number in that system.

Name	Base	No. of Digits	Digits used in the Number System	Examples
Binary	2	2	0, 1	1001_2, 111_2
Octal	8	8	0, 1, 2, 3, 4, 5, 6, 7	237_8, 3205_8
Decimal	10	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	249_{10}, 505_{10}
Hexadecimal	16	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F	$3AE6_{16}$, 25_{16}

In a base-n representation of a number, no digit exceeds n-1

Writing the base as a subscript to the number indicates the base of a number system. For example a binary number like 1001, which has a base of 2, is indicated as 1001_2.

Let us consider an example to find the importance of this type of number system over the non-positional one. Consider the number 4737 written using the decimal number system as:

Th	H	T	U
4	7	3	7

The above representation of the decimal number has 7 under the **U**nit's place, 3 under the **T**en's place, another 7 under the **H**undred's place and 4 under the **T**housand's place. It is basically a short hand representation of the value:

$$4737 = 4x10^3 + 7x10^2 + 3x10^1 + 7x10^0$$
$$= 4x1000 + 7x100 + 3x10 + 7x1$$
$$= 4000 + 700 + 30 + 7$$

Therefore each position has basically a value equal to the base of the number system raised to the power equal to the position number starting from 0, 1, 2, 3 etc. from the right as shown below:

10^3	10^2	10^1	10^0
4	7	3	7

Using the above representation two **numbers can be added very easily** as shown below:

10^3	10^2	10^1	10^0
4	7	3	7
+ 3	2	5	1
7	9	8	8

In carrying out the addition what we have done is basically we have added the digits that are in the same position i.e. digits that have the same weight as shown below:

$$4737 + 3251 = (4x10^3 + 7x10^2 + 3x10^1 + 7x10^0) + (3x10^3 + 2x10^2 + 5x10^1 + 1x10^0)$$
$$= (4+3)x10^3 + (7+2)x10^2 + (3+5)x10^1 + (7+1)x10^0$$
$$= 7x1000 + 9x100 + 8x10 + 8x1$$
$$= 7000 + 900 + 80 + 8$$
$$= 7988$$

Such type of additions is not possible in a non-positional number system. For example there is no such straight forward method of adding **III** to **IX** to get **XII**.

Note that the two 8's in the above sum have two different meanings. While the first 8 from the left has the value equal to $8x10^1=80$, the value of the second 8 is $8x10^0=8$. Hence the **position of the digit** 8 determines its value.

Count	Dec.	Bin.	Oct.	Hex
●	1	1	1	1
●●	2	10	2	2
●●●	3	11	3	3
●●●●	4	100	4	4
●●●●●	5	101	5	5
●●●●●●	6	110	6	6
●●●●●●●	7	111	7	7
●●●●●●●●	8	1000	10	8
●●●●●●●●●	9	1001	11	9
●●●●●●●●●●	10	1010	12	A
●●●●●●●●●●●	11	1011	13	B
●●●●●●●●●●●●	12	1100	14	C
●●●●●●●●●●●●●	13	1101	15	D
●●●●●●●●●●●●●●	14	1110	16	E
●●●●●●●●●●●●●●●	15	1111	17	F
●●●●●●●●●●●●●●●●	16	10000	20	10

Counting in various Number Systems

The **Egyptians** used a **base 10 number system**, but there was no symbol for zero. They used seven separate symbols for one unit, one hundred, one thousand and so on.

The **Babylonians** used a sophisticated **sexagesimal** (base 60) positional number system. But all the numerals were built with only 2 symbols – a vertical, and a corner edge.

The Indian scholar **Pingala** (circa 5th to 2nd century BC) used **binary numbers** in the form of short and long syllables making it similar to Morse code.

The **concept of zero** as a number and not merely a symbol for separation is **attributed to India** where by 9th century AD practical calculations were carried out using zero, which was treated like any other number.

3.2 Conversion of Integer Values from one number system to another

Humans are used to dealing with numbers expressed in the decimal number system. However computers deal with numbers which they store internally as binary numbers. Hence there **should be a mechanism to convert numbers from one base to another**. Conversion of numbers will be done from decimal to other number systems and from other number systems to decimal. Also we will see how to convert numbers from one number system to another both of which are not in the decimal number system.



Conversion from Decimal to other bases

- Decimal to Binary, Octal and Hexadecimal Conversion (base 10 to base 2, 8, 16)

To convert a decimal number to another base, the following steps are to be carried out:

Step1: Divide the decimal number by the base of the number to which it is to be converted

Step2: Write the remainder of the division beside the quotient

Step3: Repeat the division by dividing the quotient from the previous division by the conversion base

Step4: Repeat steps 2 and 3, till the quotient of the division is '0' in Step2

Step5: Finally **write the remainders from bottom to top to get the final result** of conversion

Example-1: Convert $(37)_{10}$ to $(?)_2$

2 | 37		
2 | 18	rem **1**	↑
2 | 9	rem **0**	
2 | 4	rem **1**	
2 | 2	rem **0**	
2 | 1	rem **0**	
0	rem **1**	

Answer: $(37)_{10} = (100101)_2$

Example-2: Convert $(126)_{10}$ to $(?)_2$

2 | 126		
2 | 63	rem **0**	↑
2 | 31	rem **1**	
2 | 15	rem **1**	
2 | 7	rem **1**	
2 | 3	rem **1**	
2 | 1	rem **1**	
0	rem **1**	

Answer: $(126)_{10} = (1111110)_2$

In the first example, we have repeatedly divided the number $(37)_{10}$ by 2, till we got the value '0' in the quotient. During division the remainders were written side by side to the quotients. Finally by **writing the remainders in the order as indicated by the arrow** we get the final result of conversion.

The following examples show the conversion from **decimal to octal**. As the base of the octal number system is 8, **divide the original number or the quotients by 8** to get the result.

Example-3: Convert $(37)_{10}$ to $(?)_8$

8 | 37		
8 | 4	rem **5**	↑
0	rem **4**	

Answer: $(37)_{10} = (45)_8$

Example-4: Convert $(126)_{10}$ to $(?)_8$

8 | 126		
8 | 15	rem **6**	↑
8 | 1	rem **7**	
0	rem **1**	

Answer: $(126)_{10} = (176)_8$

Dec.	Hex.
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

The following examples show the conversion from **decimal to hexadecimal**. As the base of hexadecimal number system is 16, **divide the original number or the quotients by 16** to get the result. While copying the remainders remember to **replace the remainder values 10, 11, 12, 13, 14, and 15 by the digits A, B, C, D, E, and F respectively** of the hexadecimal number system.

Example-5: Convert $(37)_{10}$ to $(?)_{16}$

16 | 37		
16 | 2	rem **5**	↑
0	rem **2**	

Answer: $(37)_{10} = (25)_{16}$

Example-6: Convert $(126)_{10}$ to $(?)_{16}$

16 | 126			
16 | 7	rem **14**	↑	**E**
0	rem **7**		**7**

Answer: $(126)_{10} = (7E)_{16}$

Note that the **remainder 14 was converted to its hexadecimal equivalent digit i.e. E.**

Now that we know how to convert from Decimal to all the other bases namely Binary, Octal and Hexadecimal, let us now learn the reverse conversion i.e. **from other bases to decimal.**

- Binary, Octal, Hexadecimal to Decimal Conversion (base 2, 8, 16 to base 10)

To convert a number expressed in another base to decimal there are two different methods.

Method-I

Step1: Determine the column weight of each digit in the number based on the position of the digit in the number and the base of the number system from which to convert (2, 8, or 16)

Step2: Multiply the column weight obtained in step 1 by the digit in that column, to get the column value of that particular column

Step3: Sum the products or column values calculated in step 2 to get the required converted value

Decimal to Binary conversion

'The importance of the creation of the zero mark ..... is the characteristic of the Hindu race from where it sprang. No single mathematical creation has been more potent for the general on-go of intelligence and power'. – **G. B. Halsted.**

Decimal to Octal conversion

Decimal to Hexadecimal conversion

Other base to base 10 conversion Method-1

Example-7: Convert $(100101)_2$ to $(?)_{10}$

2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	1	0	1
2^5x1	2^4x0	2^3x0	2^2x1	2^1x0	2^0x1

Position of Digit; Base of Number; Column Value

Thus converted value = $2^5x1 + 2^4x0 + 2^3x0 + 2^2x1 + 2^1x0 + 2^0x1$

= $32x1 + 16x0 + 8x0 + 4x1 + 2x0 + 1x1 = 32 + 0 + 0 + 4 + 0 + 1 = \mathbf{37_{10}}$

The required **answer** is $(\mathbf{100101})_2 = (\mathbf{37})_{10}$

In 5th century AD Indian mathematician and astronomer **Aryabhatta** stated that 'from one place to the next place is ten times in value', which may be the origin of the modern decimal based place value system.

Example-8: Convert $(1111110)_2$ to $(?)_{10}$

2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	1	1	1	0
2^6x1	2^5x1	2^4x1	2^3x1	2^2x1	2^1x1	2^0x0

Thus converted value = $2^6x1 + 2^5x1 + 2^4x1 + 2^3x1 + 2^2x1 + 2^1x1 + 2^0x0$

= $64x1 + 32x1 + 16x1 + 8x1 + 4x1 + 2x1 + 1x0$

= $64 + 32 + 16 + 8 + 4 + 2 + 0 = \mathbf{126_{10}}$

The required **answer** is $(1111110)_2 = (126)_{10}$

Double Add method (Bin. to Dec.)

Method-II

The steps for the second method (also called the **Double Add Method** for binary conversion) are:

Step1: First write down the binary number, spreading out the digits

Step2: Multiply the MSB by 2

Step3: Copy result of multiplication from the previous step and add it to the next binary digit to right

Step4: Multiply the result of step 3 by 2

Step5: Repeat the steps 3 and 4, till you reach the digit to the left of the LSB

Step6: Add the product from the previous column to the LSB and stop. The sum so obtained gives the result of conversion

The Double Add method is used to convert a binary integer value to decimal value.

Remember the **first step** does not have an addition and the **last step** does not have a multiplication.

Example-9: Convert $(100101)_2$ to $(?)_{10}$

1	0	0	1	0	1
x 2	+2	+4	+8	+18	+36
2	2	4	9	18	37
	x 2	x 2	x 2	x 2	
	4	8	18	36	

No multiplication in last step

The required **answer** is $(100101)_2 = (37)_{10}$

Example-10: Convert $(1111110)_2$ to $(?)_{10}$

1	1	1	1	1	1	0
x 2	+2	+6	+14	+30	+62	+126
2	3	7	15	31	63	126
	x 2	x 2	x 2	x 2	x 2	
	6	14	30	62	126	

The required **answer** is $(1111110)_2 = (126)_{10}$

The **main advantage** of this method is, you **do not have to calculate large powers** like 2^6, 2^7 etc.

In case the number is too large and **difficult to fit in a single line**, then one can also write the number vertically and do the calculation as shown below for the same example. Remember that the **first step does not have any addition** and the **last step does not have any multiplication by 2**.

1 → 1 x 2 = 2 [No addition in first step]
1 → (1+2) x 2 = 3x2 = 6
1 → (1+6) x 2 = 7x2 = 14
1 → (1+14) x 2 = 15x2 = 30
1 → (1+30) x 2 = 31x2 = 62
1 → (1+62) x 2 = 63x2 = 126
0 → (0+126) = 126 [No multiplication by 2 in last step]

Example-11: Convert $(11001)_2$ to $(?)_{10}$

1 → 1 x 2 = 2
1 → (1+2) x 2 = 3x2 = 6
0 → (0+6) x 2 = 6x2 = 12
0 → (0+12) x 2 = 12x2 = 24
1 → (1+24) = 25

The required **answer** is $(11001)_2 = (25)_{10}$

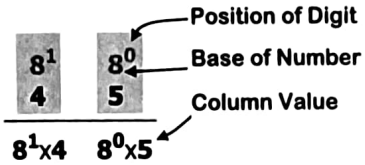
To convert a number expressed in **octal to decimal** there are again two different methods. The first method is similar to the one for binary, with the **base changed to 8** and so is the second.

Method-I



Octal to Decimal Conversion Method-I

Example-12: Convert $(45)_8$ to $(?)_{10}$



Thus converted value = $8^1x4 + 8^0x5 = 8x4 + 1x5 = 32 + 5 = \mathbf{37_{10}}$
The required **answer** is $(45)_8 = (37)_{10}$

Example-13: Convert $(176)_8$ to $(?)_{10}$

8^2	8^1	8^0
1	7	6
8^2x1	8^1x7	8^0x6

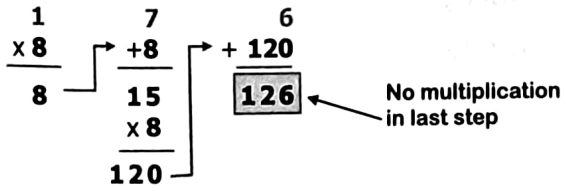
Thus converted value = $8^2x1 + 8^1x7 + 8^0x6 = 64x1 + 8x7 + 1x6 = 64 + 56 + 6 = \mathbf{126_{10}}$
The required **answer** is $(176)_8 = (126)_{10}$

Method-II



Octal to Decimal Conversion Method-2

Example-14: Convert $(176)_8$ to $(?)_{10}$



The required **answer** is $(176)_8 = (126)_{10}$

Example-15: Convert $(13254)_8$ to $(?)_{10}$

1	3	2	5	4
x 8	+8	+88	+720	+5800
8	11	90	725	5804
	x 8	x 8	x 8	
	88	720	5800	

The required **answer** is $(13254)_8 = (5804)_{10}$

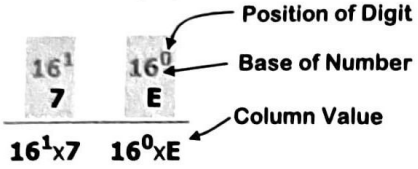
To convert a number expressed in **hexadecimal to decimal** there are again two different methods. The first method is similar to that for binary, with the **base changed to 16** and so is the second.



Hex to Decimal conversion Method-I

Method-I

Example-16: Convert $(7E)_{16}$ to $(?)_{10}$



Thus **converted** value = $16^1x7 + 16^0xE = 16x7 + 1x14 = 112 + 14 = \mathbf{126_{10}}$

The required **answer** is $(7E)_{16} = (126)_{10}$

Example-17: Convert $(9C5A)_{16}$ to $(?)_{10}$

16^3	16^2	16^1	16^0
9	C	5	A
16^3x9	16^2xC	16^1x5	16^0xA

Thus **converted** value = $16^3x9 + 16^2xC + 16^1x5 + 16^0xA$

$= 4096x9 + 256x12 + 16x5 + 1x10$

$= 36864 + 3072 + 80 + 10$

$= 40026_{10}$

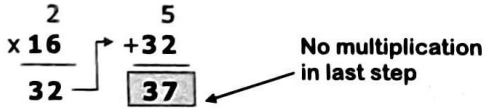
The required **answer** is $(9C5A)_{16} = (40026)_{10}$



Hex to Decimal conversion Method-II

Method-II

Example-18: Convert $(25)_{16}$ to $(?)_{10}$



The required **answer** is $(25)_{16} = (37)_{10}$

Example-19: Convert $(9C5A)_{16}$ to $(?)_{10}$

9	12	5	10
x 16	+144	+2496	+40016
144	156	2501	40026
	x 16	x 16	
	2496	40016	

The required **answer** is $(9C5A)_{16} = (40026)_{10}$

Note that the digits **C** *and* **A** *in the number have been* **written as 12 and 10** *during the conversion to decimal, as 12 represents* **C** *in decimal and 10 represents* **A** *in decimal.*

- **Binary to Octal & Octal to Binary Conversion**

It is much easier to convert between bases which are in powers of 2, i.e. binary (2^0), Octal (2^3), and hexadecimal (2^4). To convert a value from binary to octal, refer to the table to the right. Note that each octal digit (0 to 7) is represented by 3 binary digits ($0 \rightarrow 000$, $1 \rightarrow 001$, $2 \rightarrow 010$, ... $7 \rightarrow 111$). To convert a binary number to octal:

Dec.	Binary	Octal
0	000	0
1	001	1
2	010	2
3	011	3
4	100	4
5	101	5
6	110	6
7	111	7

Binary to Octal and vise versa

Step1: Group the binary digits in **groups of three from the right**

Step2: Replace each group by the corresponding octal digit

Example-20: Convert $(10011111)_2$ to $(?)_8$

$$\underbrace{0\ 1\ 0}_{2}\ \underbrace{0\ 1\ 1}_{3}\ \underbrace{1\ 1\ 1}_{7}$$

The required **answer** is $(10011111)_2 = (237)_8$

(**Note**: An extra '**0**' is put to the left of the binary number to make a complete group of 3 digits)

Three binary digits can be grouped into a single octal digit

Example-21: Convert $(110101)_2$ to $(?)_8$

$$\underbrace{1\ 1\ 0}_{6}\ \underbrace{1\ 0\ 1}_{5}$$

The required **answer** is $(110101)_2 = (65)_8$

Another method is there for the conversion. First convert the binary number to a decimal number and then convert the decimal number to octal as shown below for the previous example:

Example-22: Convert $(110101)_2$ to $(?)_8$

Step1: First convert the Binary number to Decimal

2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	0	1

Converted value $= 2^5x1 + 2^4x1 + 2^3x0 + 2^2x1 + 2^1x0 + 2^0x1$

$= 32x1 + 16x1 + 8x0 + 4x1 + 2x0 + 1x1 = 32 + 16 + 0 + 4 + 0 + 1 = \mathbf{53}_{10}$

Step2: Next convert the Decimal number to Octal

8 | 53		
8 | 6	rem **5**	↑
0	rem **6**	|

The required **answer** is $(110101)_2 = (65)_8$

We find that using either of the methods we arrive at the same answer. However to use the first method we have to **remember the octal digit corresponding to each of the binary combinations**. One way to remember this is by noting that the value of all these octal digits can be obtained by **proper combinations of the digits 4, 2, & 1**. The presence of a '1' in the binary combination should be replaced by one of the digits 4, 2, or 1 depending upon the position of the '1'. The first digit to the left corresponds to 4, followed by 2 and 1 to the right. No value is to be taken for a '0' in the binary combination. The following example will make the process clear:

4	2	1	
1	1	0	$\rightarrow 1 \times 4 + 1 \times 2 + 0 \times 1 = 4 + 2 + 0 = 6$
0	1	0	$\rightarrow 0 \times 4 + 1 \times 2 + 0 \times 1 = 0 + 2 + 0 = 2$
0	0	1	$\rightarrow 0 \times 4 + 0 \times 2 + 1 \times 1 = 0 + 0 + 1 = 1$

Converting an octal number to its binary form is just the reverse process i.e. simply **replacing each octal digit by its corresponding binary combination** as shown in the next page.

Example-23: Convert $(237)_8$ to $(?)_2$

2	3	7
0 1 0	0 1 1	1 1 1

The required **answer** is $(237)_8 = (10011111)_2$

(Note that we have removed the **extra '0'** to the left of the binary number to get the final answer)

Example-24: Convert $(603)_8$ to $(?)_2$

6	0	3
1 1 0	0 0 0	0 1 1

The required **answer** is $(603)_8 = (110000011)_2$

(Note that we have put **three '0's for the binary number that corresponds to the '0'** in the o number as each octal digit is represented by three binary digits). Moreover use the same technique discussed above to get the binary combination for a particular octal digit. Thus now **represent each oc digit as a combination of 4, 2, and 1** and **put '1' for the digits that are present and '0' for t digits that are absent**. For example:

5 = 4 + 0 + 1 → 1 0 1
7 = 4 + 2 + 1 → 1 1 1
1 = 0 + 0 + 1 → 0 0 1

Binary to Hex Conversion

- **Binary to Hex and Hex to Binary Conversion**

To convert a value from binary to hexadecimal, refer to the table to the right. Each hexadecimal digit (0 to F) can be represented by 4 binary digits (0 → 0000, 1 → 0001, 2 → 0010, ... E → 1110, F → 1111).

Step1: Group the binary digits in **groups of four from the right**

Step2: Replace each group by the corresponding hexadecimal digit

Dec.	Binary	He
0	0000	
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Example-25: Convert $(10010101)_2$ to $(?)_{16}$

1 0 0 1	0 1 0 1
9	5

The required **answer** is $(10010101)_2 = (95)_{16}$

Four binary digits can be combined to form a single hexadecimal digit.

Example-26: Convert $(1010111101)_2$ to $(?)_{16}$

0 0 1 0	1 0 1 1	1 1 0 1
2	B	D

The required **answer** is $(1010111101)_2 = (2BD)_{16}$

(Note that we have **put two extra '0**'s to the left of the binary number to make a group of 4 digits).

Like the process shown in the previous section for converting binary to octal, this conversion can also be don by first converting the binary number to decimal and then converting the decimal number to hexadecimal.

However to use the first method we have to remember the hexadecimal digit corresponding to each of th binary combinations. The value of all these hexadecimal digits can now be obtained by **proper combinations of the digits 8, 4, 2, & 1.**

The presence of a '1' in the binary combination should be replaced by one of the digits 8, 4, 2, or 1 dependin upon the position of the '1'. The **first binary digit on the left corresponds to 8**, followed by **4**, **2** and **1** a we move right. No value is to be taken for a '0' in the binary combination. For example:

8	4	2	1		
0	1	1	0	→ 0x**8** + 1x**4** + 1x**2** + 0x**1** = 0 + 4 + 2 + 0 = 6	*(as 6_{10} is 6 in hexadecimal)*
1	1	0	0	→ 1x**8** + 1x**4** + 0x**2** + 0x**1** = 8 + 4 + 0 + 0 = 12 = C	*(as 12_{10} is C in hexadecimal)*
1	0	1	1	→ 1x**8** + 0x**4** + 1x**2** + 1x**1** = 8 + 0 + 2 + 1 = 11 = B	*(as 11_{10} is B in hexadecimal)*

To convert a **hexadecimal number to binary,** simply apply the reverse process i.e. **replace each hexadecimal digit by its corresponding binary combination** as shown below:

Hexadecimal to Binary conversion

Example-27: Convert $(FADE)_{16}$ to $(?)_2$

F	A	D	E
1111	1010	1101	1110

The required **answer** is $(FADE)_{16} = (1111101011011110)_2$

Example-28: Convert $(9B0)_{16}$ to $(?)_2$

9	B	0
1001	1011	0000

The required **answer** is $(9B0)_{16} = (100110110000)_2$

(**Note** that we have put **four '0's** for the binary number that corresponds to the '0' in the hexadecimal number as each hexadecimal digit is represented by four binary digits).

This conversion can also be done by first converting the hexadecimal number to decimal and then the decimal number to a binary number. The result would have been the same in either way.

Use the **same technique** as discussed earlier to get the binary combination for a particular hexadecimal digit. Now **represent each hexadecimal digit as a combination of 8, 4, 2, and 1** and put '1' for the digits that are present and '0' for the digits that are absent. For example:

5 = 0 + 4 + 0 + 1 → 0 1 0 1
B = 11 = 8 + 0 + 2 + 1 → 1 0 1 1
E = 14 = 8 + 4 + 2 + 0 → 1 1 1 0

- **Hexadecimal to Octal & Octal to Hexadecimal Conversion**

Hexadecimal to Octal conversion

To convert from Hexadecimal to Octal and vice versa, first convert the number to binary. Then group the binary digits as per requirement to get the digits in the other number system.

Example-29: Convert $(9A8C)_{16}$ to $(?)_8$

9	A	8	C
0 0 1 0 0 1	1 0 1 0	1 0 0 0	1 1 0 0
1	1	5	2	1	4

Step1: Each Hex digit is converted to its equivalent 4 digit binary value

Step2: Each group of 3 binary digits is converted to its equivalent Octal digit

The required **answer** is $(9A8C)_{16} = (115214)_8$

In the above example, we have first written the binary equivalent of the hexadecimal number by **replacing each hexadecimal digit by its 4 bit binary equivalent**. Next we have grouped the binary number so formed, in **groups of 3 digits from the right**. We have added two 0's to the left of the binary number to make a group of 3. Finally we have **replaced each group of 3 binary digits by their octal equivalent.**

Example-30: Convert $(60D)_{16}$ to $(?)_8$

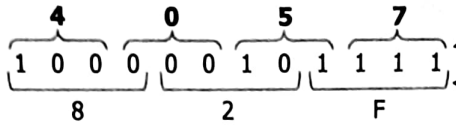
6	0	D
0 1 1 0	0 0 0	0 0 1 1 0 1
3	0	1	5

The required **answer** is $(60D)_{16} = (3015)_8$

Next we convert an octal number to its hexadecimal equivalent. **The process is the reverse of the previous process**. First write the binary equivalent of the number and then derive the hexadecimal number by grouping four binary digits **from the right**.

Dec.	Binary	Hex	Oct
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17
16	10000	10	20

Example-31: Convert $(4057)_8$ to $(?)_{16}$

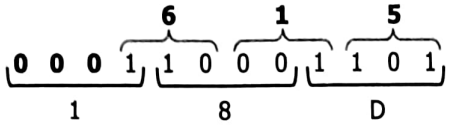


Step1: Each Octal digit is converted to its equivalent 3 digit binary value

Step2: Each group of 4 binary digits is converted to their equivalent Hexadecimal digit

The required **answer** is $(4057)_8 = (82F)_{16}$

Example-32: Convert $(615)_8$ to $(?)_{16}$



The required **answer** is $(615)_8 = (18D)_{16}$

Note that 3 extra zeroes have been added to the left of the converted binary number (without changing its value) to make a group of 4 binary digits possible, for the hexadecimal number.

The conversion from hexadecimal digit to binary and from binary to octal digit can be done easily if you remember the 8, 4, 2, 1 and 4, 2, 1 rules as discussed earlier.

3.3 Conversion of Fractional Values from one number system to another

In this section we will learn how to convert numbers from one base to another, which are **fractions**. Like the previous section, we will first convert from decimal to other number systems and vice versa. Next we will convert numbers in bases other than the decimal number system.

Decimal to Binary fraction

- **Decimal fractions to Binary fractions (base 10 to base 2)**

To convert a decimal fraction to another base, the following steps are to be carried out:

Step1: Multiply the decimal number by the base of the number to which it is converted

Step2: Write the integer part of the result of multiplication beside the product

Step3: Repeat the multiplication by multiplying the fractional part from the previous multiplication by the base of the number to which it is converted

Step4: Repeat steps 2 and 3, till the fractional part of the product of the multiplication is '0' in Step2 or you have got sufficient number of product terms

Step5: Finally **write the integer portions from top to bottom to get the final result**

Example-33: Convert $(0.236)_{10}$ to $(?)_2$

$0.236 \times 2 = \mathbf{0}.472 \quad \mathbf{0}$

$0.472 \times 2 = \mathbf{0}.994 \quad \mathbf{0}$

$0.994 \times 2 = \mathbf{1}.888 \quad \mathbf{1}$

$0.888 \times 2 = \mathbf{1}.776 \quad \mathbf{1}$

$0.776 \times 2 = \mathbf{1}.552 \quad \mathbf{1}$

$0.552 \times 2 = \mathbf{1}.104 \quad \mathbf{1}$

The required **answer** is $(0.236)_{10} = (0.001111)_2$

In the first example, we have repeatedly multiplied the number $(0.236)_{10}$ by 2, till we got sufficient number of product terms (usually 6 product terms). After multiplication the integer part of the results (viz. 0, 0, 1, 1, 1, 1) was written side by side to the products. The fraction part of a product was taken and multiplied again in the next step. Finally by ***writing the integer parts in order as indicated by the arrow*** *we got the final result of conversion.*

Example-34: Convert $(0.125)_{10}$ to $(?)_2$

$0.125 \times 2 = \mathbf{0}.250 \quad \mathbf{0}$

$0.250 \times 2 = \mathbf{0}.500 \quad \mathbf{0}$

$0.500 \times 2 = \mathbf{1}.000 \quad \mathbf{1}$

The required **answer** is $(0.125)_{10} = (0.001)_2$

- **Decimal fractions to Octal fractions (base 10 to base 8)**

Decimal to Octal Fraction

The following examples show the conversion from decimal to octal. As the base of the octal number system is 8, **multiply the original number or the fractions by 8** to get the result.

Example-35: Convert $(0.236)_{10}$ to $(?)_8$

0.236 x 8 = 1.888 **1**
0.888 x 8 = 7.104 **7**
0.104 x 8 = 0.832 **0**
0.832 x 8 = 6.656 **6**

The required **answer** is $(0.236)_{10} = (0.1706)_8$

Example-36: Convert $(0.613)_{10}$ to $(?)_8$

0.613 x 8 = 4.904 **4**
0.904 x 8 = 7.232 **7**
0.232 x 8 = 1.856 **1**

The required **answer** is $(0.613)_{10} = (0.471)_8$

- **Decimal fractions to Hexadecimal fractions (base 10 to base 16)**

Decimal to Hexadecimal Fraction

The following examples show the conversion from decimal to hexadecimal. As the base of hexadecimal number system is 16, **multiply the original number or the products by 16** to get the result.

While copying the integer part remember to **replace the integer values 10, 11, 12, 13, 14, and 15 by the digits A, B, C, D, E, and F respectively** of the hexadecimal number system.

Example-37: Convert $(0.236)_{10}$ to $(?)_{16}$

0.236 x 16 = 3.776 **3**
0.776 x 16 = 12.416 **C**
0.416 x 16 = 6.656 **6**

The required **answer** is $(0.236)_{10} = (0.3C6)_{16}$

Example-38: Convert $(0.613)_{10}$ to $(?)_{16}$

0.613 x 16 = 9.808 **9**
0.808 x 16 = 12.928 **C**
0.232 x 16 = 14.848 **E**

The required **answer** is $(0.613)_{10} = (0.9CE)_{16}$

Note that the **product integer part 12 was converted to its hexadecimal equivalent digit i.e. C** and the integer part 14 were converted to its hexadecimal equivalent E.

- **Binary fraction to Decimal fraction (base 2 to base 10)**

Binary to Decimal Fraction Method I

To convert a fraction expressed in another base to decimal, there are two different methods.

Method-I

Step1: Determine the column weight of each digit in the fraction, based on the position of the digit in the fraction and the base of the number system from which to convert (2, 8, or 16)

Step2: Multiply the column weight obtained in step 1 by the digit in that column, to get the column value of that particular column

Step3: Sum the products or column values calculated in step 2 to get the required converted value

Example-39: Convert $(0.100101)_2$ to $(?)_{10}$

	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}
0 .	1	0	0	1	0	1
	$2^{-1}x1$	$2^{-2}x0$	$2^{-3}x0$	$2^{-4}x1$	$2^{-5}x0$	$2^{-6}x1$

Position of Digit
Base of Number
Column Value

Thus converted value $= 2^{-1}x1 + 2^{-2}x0 + 2^{-3}x0 + 2^{-4}x1 + 2^{-5}x0 + 2^{-6}x1$ [**Note**: $2^{-p} = 1/2^p$]

$= (1/2^1)x1 + (1/2^2)x0 + (1/2^3)x0 + (1/2^4)x1 + (1/2^5)x0 + (1/2^6)x1$

$= (1/2)x1 + (1/4)x0 + (1/8)x0 + (1/16)x1 + (1/32)x0 + (1/64)x1$

$= (32 + 0 + 0 + 4 + 0 + 1) / 64$

$= 37/64 = (0.578125)_{10}$

Example-40: Convert $(0.0111111)_2$ to $(?)_{10}$

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}
.0	1	1	1	1	1	1
$2^{-1}x0$	$2^{-2}x1$	$2^{-3}x1$	$2^{-4}x1$	$2^{-5}x1$	$2^{-6}x1$	$2^{-7}x1$

Thus converted value $= 2^{-1}x0 + 2^{-2}x1 + 2^{-3}x1 + 2^{-4}x1 + 2^{-5}x1 + 2^{-6}x1 + 2^{-7}x1$

$= (1/2)x0 + (1/4)x1 + (1/8)x1 + (1/16)x1 + (1/32)x1 + (1/64)x1 + (1/128)x1$

$= (0 + 32 + 16 + 8 + 4 + 2 + 1)/128 = 63/128 = (0.49218)_{10}$

Half Add method (Bin. to Dec.)

Method-II

The second method is called the **Half Add Method** (for binary conversion). The steps are:

Step1: First write down the binary number, spreading out the digits

Step2: Multiply the rightmost digit by ½ i.e. 0.5

Step3: Copy the result of multiplication from the previous step and add it to the binary digit to the left

Step4: Multiply the result of step 3 by 0.5

Step5: Repeat the steps 3 and 4, till you reach the digit to the right of the binary point

Step6: The sum so obtained gives the result of conversion

Remember that **the first step (at the right) does not have any addition**.

The Half Add method is used to convert binary fractions to decimal fractions.

Example-41: Convert $(0.100101)_2$ to $(?)_{10}$

Start from here

. 1	0	0	1	0	1
+ . 15625	+ .3125	+ .625	+ .25	+ .5	x .5
1. 15625	. 3125	. 625	1. 25	. 5	. 5
x . 5	x . 5	x . 5	x . 5	x . 5	
0.578125	. 15625	. 3125	. 625	. 25	

The required **answer** is $(0.100101)_2 = (0.578125)_{10}$

Example-42: Convert $(0.0111111)_2$ to $(?)_{10}$

. 0	1	1	1	1	1	1
+. 984375	+. 96875	+. 9375	+. 875	+. 75	+. 5	x . 5
. 984375	1. 96875	1. 9375	1. 875	1. 75	1. 5	. 5
x . 5	x . 5	x . 5	x . 5	x . 5	x . 5	
. 4921875	. 984375	. 96875	. 9375	. 875	. 75	

The required **answer** is $(0.0111111)_2 = (0.4921875)_{10}$

The **main advantage** of this method is you **do not have to calculate large powers** like 2^{-6}, 2^{-7} etc.

Octal to Decimal Faction

- **Octal fraction to Decimal fraction (base 8 to base 10)**

To convert a fractional number expressed in **octal to its decimal equivalent**, a method similar to the on for binary, with the **base changed to 8** can be used.

Example-43: Convert $(0.346)_8$ to $(?)_{10}$

8^{-1}	8^{-2}	8^{-3}
. 3	4	6
$8^{-1}x3$	$8^{-2}x4$	$8^{-3}x6$

Position of Digit

Base of Number

Column Value

Thus converted value $= 8^{-1}x3 + 8^{-2}x4 + 8^{-3}x6$

$= (1/8) \times 3 + (1/64) \times 4 + (1/512) \times 6$

$= (64 \times 3 + 8 \times 4 + 1 \times 6) / 512 = (192 + 32 + 6) / 512 = 230/512 = \mathbf{0.4492}$

The required **answer** is $(0.346)_8 = (0.4492)_{10}$

Example-44: Convert $(0.7041)_8$ to $(?)_{10}$

8^{-1}	8^{-2}	8^{-3}	8^{-4}
. 7	0	4	1
$8^{-1}x7$	$8^{-2}x0$	$8^{-3}x4$	$8^{-4}x1$

Thus converted value = $8^{-1}x7 + 8^{-2}x0 + 8^{-3}x4 + 8^{-4}x1$

= $(1/8) \times 7 + (1/64) \times 0 + (1/512) \times 4 + (1/4096) \times 1$

= $(512x7 + 64x0 + 8x4 + 1x1) / 4096 = (3584 + 0 + 32 + 1) / 4096$

= $3617/4096$ = **0.8830**

The required **answer** is $(0.7041)_8 = (0.8830)_{10}$

- **Hexadecimal fraction to Decimal fraction (base 16 to base 10)**



Decimal to Octal Fraction

To convert a fractional number expressed in **hexadecimal to its decimal** equivalent, a method similar to that for binary, with the **base changed to 16** can be used.

Example-45: Convert $(0.A3C)_{16}$ to $(?)_{10}$

16^{-1}	16^{-2}	16^{-3}	← Position of Digit
. A	3	C	← Base of Number
$16^{-1}xA$	$16^{-2}x3$	$16^{-3}xC$	← Column Value

Thus converted value = $16^{-1} \times A + 16^{-2} \times 3 + 16^{-3} \times C$

= $(1/16) \times 10 + (1/256) \times 3 + (1/4096) \times 12$ [*As Hex. A=10, C=12*]

= $(256 \times 10 + 16 \times 3 + 1 \times 12) / 4096 = (2560 + 48 + 12) / 4096$

= $2620/4096$ = **0.63965**

The required **answer** is $(0.A3C)_{16} = (0.63965)_{10}$

- **Binary to Octal & Octal to Binary Fraction Conversion**



Binary to Octal Fraction

To convert a value from binary to octal, refer to the table to the right. Note that each octal digit (0 to 7) is represented by 3 binary digits (0 → 000, 1 → 001, 2 → 010, ... 7 → 111). To convert a **binary fraction to octal**:

Dec.	Binary	Octal
0	000	0
1	001	1
2	010	2
3	011	3
4	100	4
5	101	5
6	110	6
7	111	7

Step1: Group the binary digits in **groups of three from the left**

Step2: Replace each group by the corresponding octal digit

Example-46: Convert $(0.10011111)_2$ to $(?)_8$

. 1 0 0 | 1 1 1 | 1 1 0

4 7 6

The required **answer** is $(0.10011111)_2 = (0.476)_8$

(**Note**: An extra '**0**' is put to the right of the binary fraction to make a complete group of 3 digits)

Example-47: Convert $(0.110101)_2$ to $(?)_8$

. 1 1 0 | 1 0 1

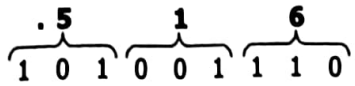
6 5

The required **answer** is $(0.110101)_2 = (0.65)_8$

Another method is there for the conversion. First convert the binary fraction to a decimal fraction and then convert the decimal fraction to octal. Both methods will give you the same result.

Converting an **octal fraction to its binary form** is just the reverse process i.e. simply **replace each octal digit by its corresponding binary combination** as shown in the next page.

Example-48: Convert $(0.516)_8$ to $(?)_2$



The required **answer** is $(0.516)_8 = (0.101001110)_2$

Dec.	Binary	Hexa.	Oct.
0	0 0 0 0	0	0
1	0 0 0 1	1	1
2	0 0 1 0	2	2
3	0 0 1 1	3	3
4	0 1 0 0	4	4
5	0 1 0 1	5	5
6	0 1 1 0	6	6
7	0 1 1 1	7	7
8	1 0 0 0	8	10
9	1 0 0 1	9	11
10	1 0 1 0	A	12
11	1 0 1 1	B	13
12	1 1 0 0	C	14
13	1 1 0 1	D	15
14	1 1 1 0	E	16
15	1 1 1 1	F	17

Binary to Hex Fractions

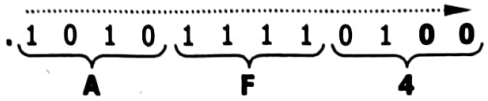
- **Binary to Hexadecimal and Hexadecimal to Binary Conversion**

To convert a fractional value from binary to hexadecimal, refer to the table on the right.

Step1: Group the binary digits in **groups of four from the left**

Step2: Replace each group by the corresponding hex. digit

Example-49: Convert $(0.1010111101)_2$ to $(?)_{16}$

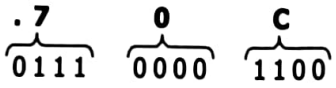


The required **answer** is $(0.1010111101)_2 = (0.AF4)_{16}$

(Note that in this case we have put two extra '**0**'s to the **right of the binary fractional number** to make group of 4 digits).

To convert a hexadecimal fraction to binary, simply apply the reverse process i.e. **replace each hexadecimal digit by its corresponding binary combination** as shown below:

Example-50: Convert $(0.70C)_{16}$ to $(?)_2$



The required **answer** is $(0.70C)_{16} = (0.011100011)_2$

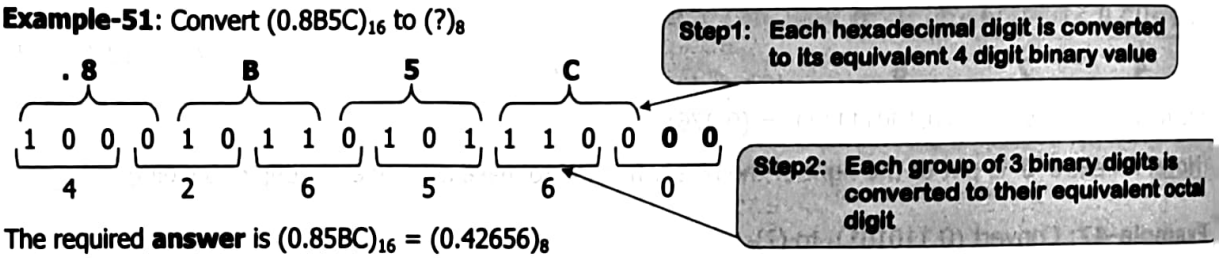
Note that we have put **four '0's** for the binary number that corresponds to the '0' in the hexadecimal number as each hexadecimal digit is represented by four binary digits. Moreover we have not written the trailing '0's at the end.

Hex to Octal Fraction

- **Hexadecimal to Octal & Octal to Hexadecimal Conversion**

To convert from Hexadecimal to Octal and vice versa, first convert the fraction to binary. Then **group the binary digits from left** as per requirement to get the digits in the other number system.

Example-51: Convert $(0.8B5C)_{16}$ to $(?)_8$

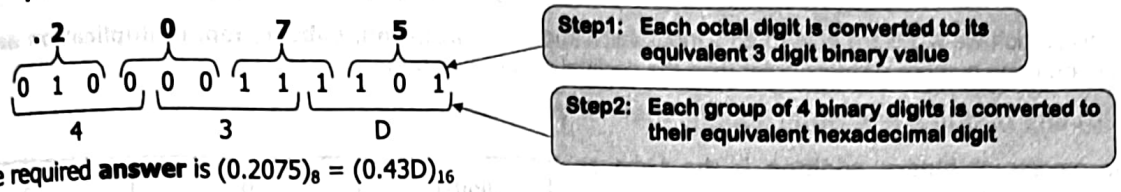


The required **answer** is $(0.85BC)_{16} = (0.42656)_8$

In the above example, we have first written the binary equivalent of the hexadecimal fraction by **replacing each hexadecimal digit by its 4 bit binary equivalent**. Next we have grouped the binary number so formed, in **groups of 3 digits from the left** (since the number is a fraction). We have added two 0's to the right of the binary number to make a group of 3. Finally we have replaced each group of 3 binary digits by its octal equivalent digit.

Next we convert an **octal fraction to its hexadecimal equivalent**. First write the binary equivalent of the number and then derive the hexadecimal number by grouping four binary digits from left.

Example-52: Convert $(0.2075)_8$ to $(?)_{16}$



The required **answer** is $(0.2075)_8 = (0.43D)_{16}$

- Converting a number from any base to any other base

Keeping in mind what you have learnt so far it is very easy to convert from any base to any other base. The steps to do the conversion are shown below for converting a number in base 6 to base 9.

Conversion from any base to any other base

Example-53: Convert $(532)_6$ to $(?)_9$

Step1: First convert the number to decimal by taking the column value of each digit in base 6

6^2	6^1	6^0
5	3	2

Base of number system FROM which to convert

Converted value = $6^2 \times 5 + 6^1 \times 3 + 6^0 \times 2$

= $36 \times 5 + 6 \times 3 + 1 \times 2 = 180 + 18 + 2 = \mathbf{200}_{10}$

Step2: Next convert the decimal number to base 9

9 | 200		
9 | 22	rem	2
9 | 2	rem	4
0	rem	2

Base of number system TO which to convert

The required **answer** is $(532)_6 = (242)_9$

Example-54: Find the missing number y in: $(255)_8 + (y)_2 = (135)_{10} + (3D)_{16}$

In the above problem, we have to **first convert all the numbers to a common base** and then do the calculations. Finally the result can be displayed in the required base by suitable conversions. In our example let us **convert all the numbers to decimal** to make the calculations easy.

$(255)_8 = (?)_{10}$

8^2	8^1	8^0
2	5	5

Thus converted value = $8^2x2 + 8^1x5 + 8^0x5 = 64x2 + 8x5 + 1x5 = 128 + 40 + 5 = \mathbf{173}_{10}$

$(3D)_{16} = (?)_{10}$

16^1	16^0
3	D

Thus converted value = $16^1x3 + 16^0xD = 16x3 + 1x13 = 48 + 13 = \mathbf{61}_{10}$

Hence we have, $(y)_2 = (135)_{10} + (3D)_{16} - (255)_8$

Or $(y)_2 = (135)_{10} + (61)_{10} - (173)_{10} = \mathbf{(23)}_{10}$

$(23)_{10} = (?)_2$

2 | 23		
2 | 11	rem	1
2 | 5	rem	1
2 | 2	rem	1
2 | 1	rem	0
0	rem	1

Therefore the required value of y in base 2 is **$(10111)_2$**

3.4 Arithmetic Operations in various Number Systems

In this section we will learn how to do various calculations like **addition, subtraction, multiplication and division** using number systems other than the decimal number system.

- **Binary Addition and Subtraction**

The process of **binary addition** is the same as decimal addition. However binary addition is much simpler as we have to deal with **only two digits** '0' and '1'. If we remember the binary addition table it will be very easy to perform the addition:

digit-1	0	0	1	1
digit-2	+ 0	+ 1	+ 0	+ 1
Sum	0	1	1	1 0
				Carry

Binary Addition Table



Binary Addition

Example-55: Find $(101101)_2 + (110000)_2$

$$\begin{array}{r} 1\ 0\ 1\ 1\ 0\ 1 \\ +\ 1\ 1\ 0\ 0\ 0\ 0 \\ \hline 1\ 0\ 1\ 1\ 1\ 0\ 1 \end{array}$$

[Using the above four results to get the total sum]

The required **sum** is $(1011101)_2$

Example-56: Find $(1101)_2 + (1011)_2$. Let us do this addition step by step, to understand the process.

Column-1

$$\begin{array}{r} \overset{\downarrow}{1} \\ 1\ 1\ 0\ 1 \\ +\ 1\ 0\ 1\ 1 \\ \hline 0 \end{array}$$

Carry of 1 is forwarded to next column as 1+1 = **10**

Column sum is (1+1)=**10** ∴ digit **0** is placed under the column

Column-2

$$\begin{array}{r} \overset{\downarrow}{1}\ 1 \\ 1\ 1\ 0\ 1 \\ +\ 1\ 0\ 1\ 1 \\ \hline 0\ 0 \end{array}$$

Carry of 1 is forwarded to next column as (1+0)+1 = 1+1 = **10**

Column sum is (1+0+1)=**10** ∴ digit **0** is placed under the column

Column-3

$$\begin{array}{r} \overset{\downarrow}{1}\ 1 \\ 1\ 1\ 0\ 1 \\ +\ 1\ 0\ 1\ 1 \\ \hline 0\ 0\ 0 \end{array}$$

Carry of 1 is forwarded to next column as (1+1)+0 = 10+0 = **10**

Column sum is (1+1+0)=**10** ∴ digit **0** is placed under the column

Column-4

$$\begin{array}{r} 1 \\ 1\ 1\ 0\ 1 \\ +\ 1\ 0\ 1\ 1 \\ \hline 1\ 1\ 0\ 0\ 0 \end{array}$$

Carry of 1 is forwarded to next column as (1+1)+1 = 10+1 = **11**

Column sum is (1+1+1)=**11** ∴ digit **1** is placed under the column

There are four possibilities for binary addition. These are: 0+0=0, 0+1=1, 1+0=1, 1+1=10

Note that for the addition of the digits in column 4, we have to add three 1's. First we add 1+1=10, as per the table shown above. This is the binary equivalent of decimal 2. When we add 1 to this we get 10+1=11. This is nothing but the binary equivalent of 2+1=3. Thus $1+1+1 = 3_{10} = 11_2$. As there are no further digits to add, this final carry is written as the left most bit of the sum.

Therefore the required **sum** is $(11000)_2$

Example-57: Find $(10101)_2 + (11101)_2$

$$\begin{array}{r} 1\ 1\ \ \ 1 \\ 1\ 0\ 1\ 0\ 1 \\ +\ 1\ 1\ 1\ 0\ 1 \\ \hline 1\ 1\ 0\ 0\ 1\ 0 \end{array} \quad \leftarrow \textit{Carry}$$

The required **sum** is $(110010)_2$

Example-58: Find $(11011)_2 + (10011)_2$

$$\begin{array}{r} 1 \quad\quad 1\ 1 \\ 1\ 1\ 0\ 1\ 1 \\ +\ 1\ 0\ 0\ 1\ 1 \\ \hline 2\ 1\ 1\ 3\ 2 \\ -2\ \ \ \ -2\ -2 \\ \hline 1\ 0\ 1\ 1\ 1\ 0 \end{array} \quad \begin{array}{l} \leftarrow \textit{Carry forward 1} \\ \\ \\ \leftarrow \textit{Decimal sum} \\ \leftarrow \textit{2 subtracted} \\ \leftarrow \textit{Binary sum} \end{array}$$

The required **sum** is $(101110)_2$

There is an **alternative method to find the sum**. When adding the digits along with the carry:

- If the **decimal sum** of the digits in a column is **less than 2**, then write that as the column sum.
- If the **decimal sum** of the digits is **more than or equal to 2**, then **subtract 2** from that sum and write the result as the binary column sum. Next **carry forward 1 to the next column.**

Example-58 is done using the second method.

Binary Subtraction is similar to normal subtraction. The table given on the right gives a summary of the **four different cases** that can occur when subtracting two binary digits.

digit-1	0	1	1	1
digit-2	− 0	− 1	− 0	− 1
Diff.	0	1	1	0
		Borrow		

Other than the second combination, all the other three combinations are similar to normal subtraction. In the second combination, we are trying to subtract 1 from 0 i.e. a larger number from a smaller number. To do this, as in case of normal decimal subtraction we have to borrow '1' from the next column. After the borrow operation, we have to subtract 1 from 10_2. Note that, as previously discussed, 10_2 is same as 2_{10}. **Thus when we subtract 1 from 2, we are left with (2-1)=1, and hence the result.**

Example-59: Find $(1101)_2 - (11)_2$. Let us do this subtraction step by step, to understand the process.

Column-1

```
  1  1  0  1
− 0  0  1  1
-------------
                 0
```

The column difference is (1-1) = 0

Column-2

```
  1  0  1  0  1
− 0  0  1  1
-------------
          1  0
```

Borrow of 1 is taken from the next column

The column difference is $(10_2\text{-}1_2) = 1_2$

Column-3

```
  1  0  0  1
− 0  0  1  1
-------------
       0  1  0
```

Due to the borrow taken, this position has a 0 now.

The column difference is (0-0) = 0

Column-4

```
  1  1  0  1
− 0  0  1  1
-------------
  1  0  1  0
```

The column difference is (1-0) = 1

The required **difference** is $(1010)_2$

Example-60: Find $(1001)_2 - (111)_2$. Let us do this subtraction step by step to understand the operation.

Column-1

```
  1  0  0  1
− 0  1  1  1
-------------
                 0
```

The column difference is (1-1) = 0

Column-2a

```
  0  1  1  0  0  1
− 0  1  1  1
-------------
                 0
```

Borrow of 1 taken from column4 from the value 1

Column-2b

```
  0  1  1  0  1  0  1
− 0  1  1  1
-------------
          1  0
```

Borrow of 1 taken from column3 from the value 10
[Remember $10_2\text{-}1_2=1_2$, hence 1 remains in column-3]
The final column difference is $(10\text{-}1)_2 = 1$

Column-3

```
  0  1  1  0  1  0  1
− 0     1  1  1
-------------
       0  1  0
```

The column difference is (1-1) = 0

Column-4

```
  0  1  1  0  1  0  1
− 0  1  1  1
-------------
  0  0  1  0
```

The column difference is (0-0) = 0

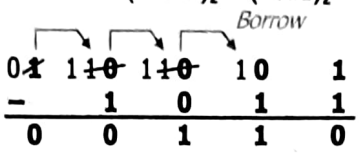
The required **difference** is $(10)_2$



Binary subtraction

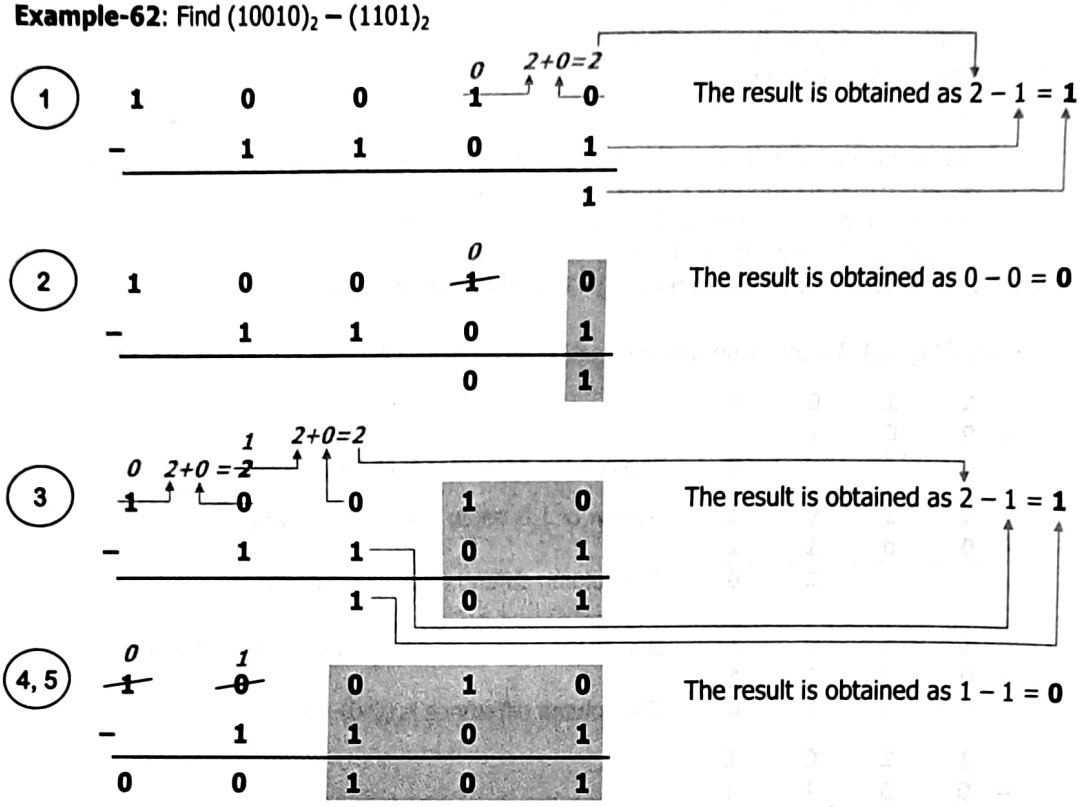
Binary subtraction has four possibilities. These are:
0-0=0, 1-0=1
1-1=0, 0-1=1

Example-61: Find $(10001)_2 - (1011)_2$



The required **difference** is $(110)_2$

Example-62: Find $(10010)_2 - (1101)_2$



Example-62 is done using a different method. In this method:

- Whenever a **borrow is made** from the next column, **reduce 1 from that column** and **write the borrowed value as 2** (base value of binary) over the current column
- **Add this 2** with the digit at the top of the current column
- Next **perform subtraction** similar to decimal subtraction in the current column

Column1 in the said example shows this situation (first 2+0=2, next the column difference 2-1=1).

However, in case the **adjacent column also has a '0'**, then:

- **Borrow from** the **next available column** and reduce 1 there
- **Move the borrowed value** towards the right up to the current column
- Over every column **in between**:
  - First **write the borrowed value as 2** and add it to the digit at the column top
  - **Reduce the new digit by 1** and move the 2 to the next column till you reach the current column
- Finally add the 2 to the digit at the top of the current column and subtract

Column3 in the above example shows this situation where the final subtraction gives 2-1=1



Octal Addition and Subtraction

- **Octal Addition and Subtraction**

While doing addition and subtraction using octal numbers, just remember that in octal number system there are only eight digits from 0 to 7. Thus after 7 we do not have 8 or 9 but 10, 11, 12 ..., up to 17. After 17 since we do not have 18, we will have 20. The **table below** gives a relationship between the first 16 decimal numbers and their octal equivalents.

Ex63 Find $(453)_8 + (312)_8$

```
  4 5 3
+ 3 1 2
  7 6 5
```

The **sum** is $(765)_8$

Ex64 Find $(5567)_8 + (4025)_8$

```
    1 1 ← Carry 1
  5 5 6 7
+ 4 0 2 5
1 1 6 1 4
```

The **sum** is $(11614)_8$

Ex65 Find $(14735)_8 + (36127)_8$

```
1← 1←    1←      ←Carry 1
1  4  7  3  5
+ 3  6  1  2  7
5 11  8  6 12   ←Decimal sum
  -8 -8    -8   ←8 subtracted
5  3  0  6  4   ←Octal sum
```

The **sum** is $(53064)_8$

In **Example-64** the first column on the right (7+5) gives 14 in base 8. This is obtained by **counting 5 digits from 7**. In octal after 7 we have 10, thus 7+5 gives 7+**1**=10, 10+**1**=11, 11+**1**=12, 12+**1**=13, 13+**1**=**14**. In the unit's place 4 is written and 1 is carried forward to the next column. In the next column, the digits are added as 6+1+2=7+2=**11 in octal**. In this way the other columns are added.

Dec.	Oct.
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	10
9	11
10	12
11	13
12	14
13	15
14	16
15	17
16	20

There is an **alternative method to find the sum** similar to the alternative method for binary addition. When adding octal digits in a column along with a carry (if any):

- If the **decimal sum** of the digits in a column is **less than 8** (base of octal), then write that as the octal column sum. **No carry forward** is generated.
- If the **decimal sum** of the digits is **greater than or equal to 8**, then **subtract 8** from that sum and write the result as the octal column sum. Next **carry forward 1 to the next column.**

Example-65 is done using the **alternative method**. The first column will give $(5+7)=12_{10}$. As 12 is greater than 8, we do the subtraction (12-8)=**4** and write it below that column, and carry forward 1 to the next column. In column2 we have $(1+3+2)=6_{10}$. No change is done as it is less than 8, and hence written as it is. In column3 we have $(7+1)=8_{10}$. As it is equal to 8, we subtract 8 to get (8-8)=**0**, and carry forward 1 to the next column. In column4 we have $(1+6+4)=11_{10}$. As it is greater than 8, we subtract 8 to get (11-8)=**3**, and carry forward 1 to the next column. In column5 we have $(1+1+3)=5_{10}$. No change is done as it is less than 8. The final result is thus $(53064)_8$.



Octal Subtraction

Octal subtraction is similar to decimal subtraction. However we have to remember that while doing the subtraction one has to count the difference in octal form. For example if one has to find the difference between 13_8 and 6_8, then count from $(6+1)=7_8$ up to 13_8 **in octal**. From the previous page chart we find that the numbers occurring from 7 to 13 are 7, 10, 11, 12, 13. Hence the total difference in count is 5. Remember that **10 comes after 7** in octal number system, therefore **do not count** as 7, 8, 9, 10, 11, 12, 13 as in the decimal number system. This would have produced the wrong result 7 (since 13 – 6 = 7 in the decimal number system).

Ex-66 Find $(7236)_8 - (3154)_8$

```
    Borrow
  7 1~~2~~ 13  6
- 3  1  5  4
  4  0  6  2
```

The **difference** is $(4062)_8$

Ex-67 Find $(5405)_8 - (77)_8$

```
      Borrow
5  3~~4~~ 7~~1~~0 15
-0    0    7    7
5    3    0    6
```

The **difference** is $(5306)_8$

In **Example-67**, in the first column on the right we have to subtract 7_8 from 5_8. Hence we have to borrow 1 from the next column, which contains 0. So we borrow 1 from the column next to it which contains 4. After borrowing 1 from that column we have 3 left there. Column2 then becomes 10_8. We next borrow 1 from the second column. Subtracting 1 from 10 leaves behind 7. Remember in octal, after 7 we have 10 and not 8. Thus when we subtract 1 from 10, we have 7 left. After the final borrow, in column1 we have 15–7. To get the difference, count 15 from 10 keeping in mind **after 7 we have 10**. From 10 to 15 the count will be 10, 11, 12, 13, 14, 15 i.e. a total of 6.

The next example i.e. example-68 is done using a different method. In this method:

- Whenever a **borrow is made** from the next column, **reduce 1 from that column** and **write the borrowed value as 8** (base value of octal) over the current column

- **Add this 8** with the digit at the top of the current column
- Next **perform subtraction** similar to decimal subtraction in the current column

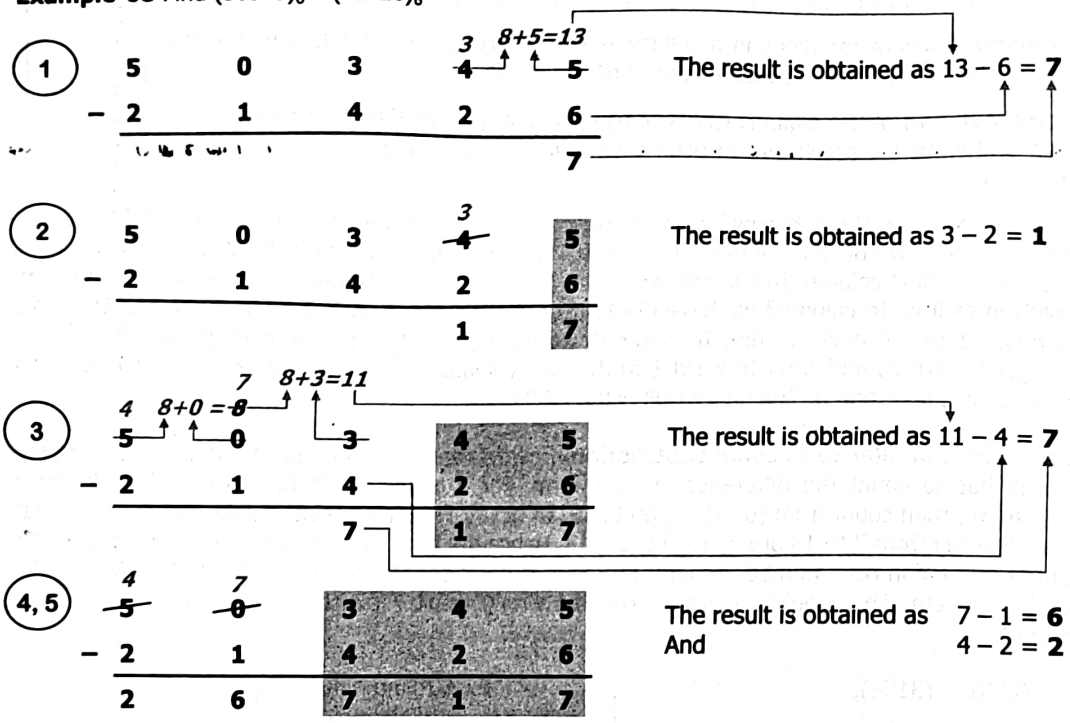
Column1 in the said example shows this situation (first 8+5=13, next the column difference 13–6=7).

However, in case the **adjacent column also has a '0'**, then:

- **Borrow from** the **next available column** and reduce 1 there
- **Move the borrowed value** towards the right up to the current column
- Over every column **in between**:
  - First **write the borrowed value as 8** and add it to the digit at the column top
  - **Reduce the new digit by 1** and move the 8 to the next column till you reach the current column
- Finally add the 8 to the digit at the top of the current column and do the subtraction

Column3 in the above example shows this situation where the final subtraction gives 11-4=7.

Example-68 Find $(50345)_8 - (21426)_8$



The **difference** is $(26717)_8$

- <u>Hexadecimal Addition and Subtraction</u>

While doing addition and subtraction using hexadecimal numbers, just remember that in hexadecimal there are sixteen digits from 0 to F. Thus **after 9 we do not have 10** but A, B, ..., up to F. After F since we do not have any extra digit, we will have 10, 11, 12,..., 1E, 1F and so on. Then after 1F we will have 20 in hex. The **table in the next page** shows the first 32 decimal numbers and their hexadecimal equivalents.

Hexadecimal Addition

Example-69: $(69)_{16}+(34)_{16}$

```
  6 9
+ 3 4
  9 D
```

(by writing D, the hex equivalent of 13)

The **sum** is $(9D)_{16}$

Example-70: $(572B)_{16}+(4968)_{16}$

```
  1   1   ← Carry
  5 7 2 B
+ 4 9 6 8
  A 0 9 3
```

The **sum** is $(A093)_{16}$

Example-71: Find $(9A5)_{16} + (94C)_{16}$

```
1←  1←   ← Carry forward
  9 A 5
+ 9 4 C
 18 15 17  ← Decimal sum
−16  −16   ← 16 subtracted
 1  2 F 1  ← Hexadecimal sum
```

The **sum** is $(12F1)_{16}$

Dec.	Hex
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F
16	10
17	11
18	12
19	13
20	14
21	15
22	16
23	17
24	18
25	19
26	1A
27	1B
28	1C
29	1D
30	1E
31	1F

In **Example-70**, note that in the first column on the right (B+8) gives 13 in base 16. The value 13 is obtained by **counting 8 digits from B**. Since in hex after B we have C, thus B+8 gives B+**1**=C, C+**1**=D, D+**1**=E, E+**1**=F, F+**1**=10, 10+**1**=11, 11+**1**=12, and finally 12+**1**=**13**.

In the unit's place 3 is written and the 1 is carried forward. In the next column, the digits are added as 1+2+6=3+6=9. In column3 we have 7+9=16 in decimal, which is equal to 10 in hex (refer to the table in the last page). The '0' is written down and the '1' is carried forward. In column4, the digits 1+5+4 are added to get the digit **A** in hexadecimal as $1+5+4=6+4=10_{10}=\mathbf{A}_{16}$.

In the **alternative method to find the hexadecimal sum** along with the carry (if any):

- If **decimal sum** of the digits in a column is **less than 16**, then write that as the hexadecimal column sum. **No carry forward** is generated. Remember if the column sum is from 10_{10} to 15_{10} then write **A** to **F** in hexadecimal.
- If the **decimal sum** of the digits is **greater than or equal to 16**, then **subtract 16** from that sum and write the result as the hex column sum. Next **carry forward 1 to the next column**.

The **Example-71** is done **using the alternative method**. When getting the decimal sum of 5+C we have $(5+C_{16})=(5+12_{10})=17_{10}$. As it is more than 16, we subtract 16 from the result and 1 is carried forward to the next column. In the next column we have $(1+A_{16}+4)=(1+10_{10}+4)=15_{10}$. As it is less than 16, hence the column sum is directly written as the hexadecimal sum. However, as 15_{10} = F in hexadecimal, we write F as the column sum. Next we have $(9+9)=18_{10}$. Being more than 16, we subtract 16 and write 2 as the column sum and carry forward 1 to the next place to get the final sum as $(12F1)_{16}$.

The following examples show how to do **hexadecimal subtraction using different methods**. Remember that with 16 digits in hex number system, we have A_{16} after 9_{16}, and 10_{16} after F_{16}.

Hexadecimal subtraction

Example-72 Find $(9C8D)_{16}-(25AB)_{16}$

Borrow

9	B	C	18	D
− 2	5	A	B	
7	6	E	2	

The **difference** is $(76E2)_{16}$

Example-73: Find $(CA07)_{16}-(2289)_{16}$

Borrow

C	9	A	F	10	17
−2	2	8	9		
A	7	7	E		

The **difference** is $(A77E)_{16}$

In **Example-73** in the first column to the right we have to subtract 9_{16} from 7_{16}. To do this we have to borrow 1 from the next column which has 0. We therefore borrow from the next column which contains A. After borrowing 1 from that column we have 9 left in that column (remember that the digit that comes after 9 is A in hex). Column2 then becomes 10. We next borrow 1 from the second column. Subtracting 1 from 10 leaves behind F (as the digit before 10 is F in hexadecimal). After the final borrow, in column1 we have 17–9. To get the difference, count 17 after 9 keeping in mind that 10 comes after F. Thus from 9 to 17 the count will be A, B, C, D, E, F, 10, 11, 12, 13, 14, 15, 16, 17 i.e. a total of 14 counts, which in hexadecimal is E. In column2 we have to subtract 8 from F. Counting up to F after 8 gives 9, A, B, C, D, E, F, i.e. a total of 7 counts. Similarly in column4 we have to subtract 2 from C. To get the result, count up to C after 2 i.e. 3, 4, 5, 6, 7, 8, 9, A, B, C. This gives a total count of 10, which in hexadecimal is A.

The next example i.e. example-74 is done using the alternative method. In this method:

- Whenever a **borrow is made** from the next column, **reduce 1 from that column** and **write the borrowed value as 16** (base value of hexadecimal) over the current column
- **Add this 16** with the digit at the top of the current column
- Next **perform subtraction** similar to decimal subtraction in the current column

Column1 in the said example shows this situation (first 16+8=24, next the column difference 24 – D = B).

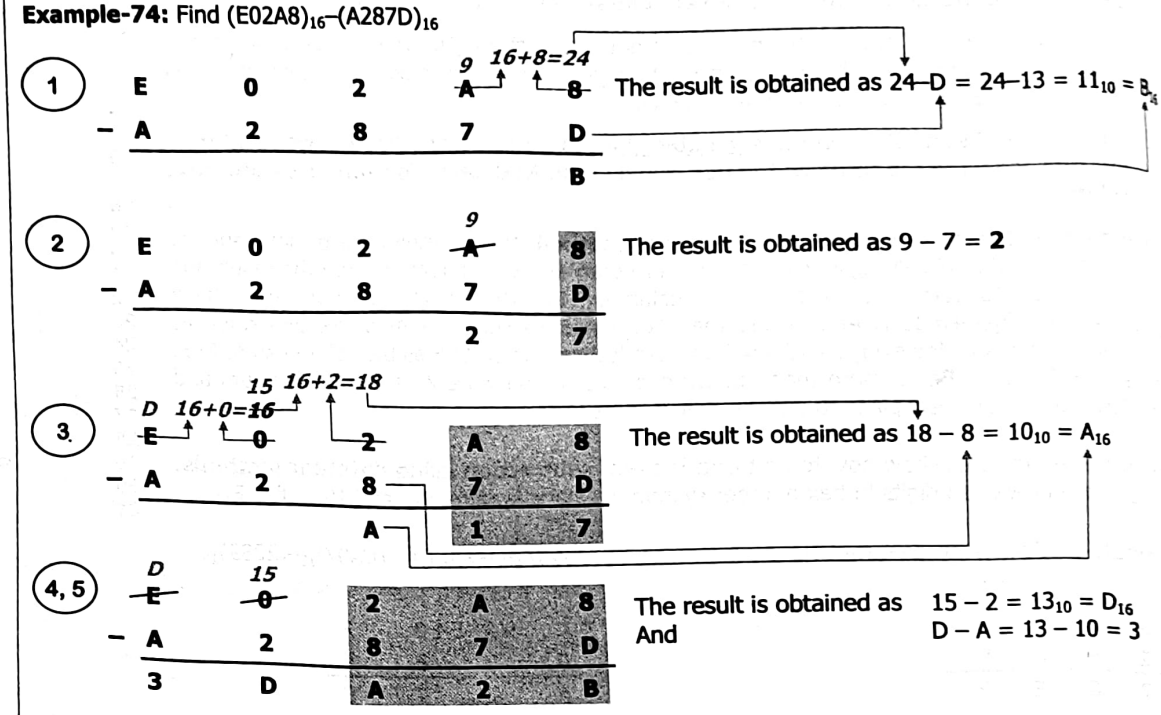
However, in case the **adjacent column also has a '0'**, then:

- **Borrow from** the **next available column** and reduce 1 there

- **Move the borrowed value** towards the right up to the current column
- Over every column **in between**:
  - First **write the borrowed value as 16** and add it to the digit at the column top
  - **Reduce the new digit by 1** and move the 16 to the next column till you reach the current column
- Finally add the 16 to the digit at the top of the current column and do the subtraction

Column3 in the above example shows this situation where the final subtraction gives 18 – 8 = A.

Example-74: Find $(E02A8)_{16}-(A287D)_{16}$



The **difference** is $(3DA2B)_{16}$

As a **third alternative method** for doing addition and subtraction in any base in general, **first convert the numbers to decimal** and then **do the calculation.** After the calculation is done and the result is obtained in decimal, **convert the result back to the original base.**

- Binary Multiplication and Division

Binary Multiplication

Example-75: Find $(10010111)_2 \times (1001)_2$

		1	0	0	1	0	1	1	1
					x	1	0	0	1
				1	1				
		1	1	0	0	1	0	1	1	1
	1	0	0	0	0	0	0	0	0	x
	0	0	0	0	0	0	0	0	x	x
1	0	0	1	0	1	1	1	x	x	x
1	0	1	0	1	0	0	1	1	1	1

Answer: The **product** is $(10101001111)_2$

Example-76: Find $(11111000)_2 \times (10.11)_2$

	1	1	1	1	1	0	0	0	
				x	1	0	.	1	1
		10	1	1					
	10	1	1	1	1	1	0	0	0
10	1	1	1	1	1	0	0	0	x
1	0	0	0	0	0	0	0	x	x
1	1	1	1	1	0	0	0	x	x	x
1	0	1	0	1	0	1	0	1	0	0	0

Answer: The **product** is $(1010101010.00)_2$

The first example is very simple. For the highlighted column we have to add 1+0+0+1 = 1+0+1 = 1+1 = 10_2. The '0' is written under that column and the '1' is carried forward to the next column, where the numbers are added along with the carry as 1+(0+0+0+1) = 1+1 = 10_2.

Dec.	Bin.
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111

For the **second multiplication**, if you can remember the first few binary numbers then it will be easy to calculate the sum. The table on the right gives a list of binary numbers from 0 to decimal 7. While doing the column sum always remember that you **get the next binary number after adding '1'** to the present binary number.

Consider the highlighted column of the second multiplication. The digits that get added are 1+1+0+0=10. The 0 is written under the column and the 1 is passed to the next column as carry (the **carry digits are shown in colour**).

In the next column along with the carry of 1 we add 1+(1+1+0+0) = 1+(10+0+0) = 1+10 = 11. The unit's place 1 is written under the column and the other 1 is carried forward to the next column. The column sum in the next column along with the carry will be 1+(1+1+0+1) = 1+(10+0+1) = 1+(10+1) = 1+11 = 100. The 0 in the unit's place is written down and the 10 passed to the next column. In the next column the column sum along with the carry will be10+(1+1+0+1)=10+(10+0+1)=10+(10+1)=10+11=101. The 1 in the unit's place is written down and 10 passed to the next column as carry.

Similar additions take place in the next columns. The final result is given after **adjusting the binary point**.

The following examples show how to **perform binary division**. The process is explained later.

Example-77: Find $(10011)_2 \div (110)_2$

```
110 | 10011 | 011
    -110↓
      111
     -110
        1
```

Answer: Quotient = 11_2, Rem. = 1_2

Example-78: Find $(11011101)_2 \div (11)_2$

```
11 | 11011101 | 1001001
   -11↓↓↓
    00011
     -11↓↓↓
      00101
       -11
        10
```

Answer: Quotient = 1001001_2, Rem. = 10_2

Binary Division

For binary division, follow the same method as in decimal number division. However remember that the **subtraction involved is binary subtraction** and do it the way shown in page P1-3-17.

In doing the first division, we first checked if 110 go into the first three digits of the dividend i.e. into 100. As it does not, we put a 0 for the quotient part. Next we checked if 110 go into 1001. It does go into 1001 once. Hence we put 1 for the quotient. We then bring down the next 1 from the dividend and check if 110 goes into 111. As it goes into 111 only once, we put a 1 for the quotient. The final remainder is 1 and the quotient is 011 i.e. simply 11_2.

In binary division the divisor will either go into the partial dividend once or will not be divisible at all.

In binary, during division while checking if the divisor goes into a part of the dividend, remember that **either it will not go** (put a '0' for the quotient part in that case), **or it will go into that part of the dividend only once** (in that case put a '1' for the quotient part). You will never have a case where the divisor goes into a part of the dividend more than once.

The next two examples show division **including a fractional part**. Put a binary point after the integer portion of the quotient when all the digits from the original number are already taken.

Example-79: Find $(11010)_2 \div (100)_2$

```
100 | 11010 | 110.1
   -100↓
    101
   -100
     100
    -100
     000
```

Put a binary point, and take down a '0', after all the digits of the original number are already taken

Answer: Quotient = 110.1_2

Example-80: Find $(110110)_2 \div (111)_2$

```
111 | 110110 | 111.1 01
   -111
    1101
   -111
     1100
    -111
     1010
    -111
     1100
      -111
       101
```

Answer: Quotient = 111.101_2 (*up to 3 places*)

Binary Division with fractions

An **alternative method for Binary Multiplication** using the concept of **partial sums** is shown below:

Example-81: Find $(1011)_2 \times (111)_2$

```
          1 0 1 1
        x 1 1 1
      -----------
    1 1 1 1
          1 0 1 1     1st partial product
      + 1 0 1 1 x     2nd partial product
      -----------
  1
    1 0 0 0 0 1       1st partial sum
  + 1 0 1 1 x x       3rd partial product
  -----------
  1 0 0 1 1 0 1       final sum
```

The example on the left is done by **adding the partial products** at each step of the multiplication process.

After the first two steps of the multiplication process, the 1st and the 2nd partial products are added to get the **first partial sum 100001** (shaded region). Next the third partial product is calculated and added to the previous partial sum to get the final product.

The **final product** is thus $(1001101)_2$

The method eliminates the need to do long additions.

Signed numbers

3.5 Representation of Signed Numbers using Complements

When we write a decimal number, we usually write the magnitude of the number preceded by a + or – sign, which indicates the sign of the numbers. In a similar manner we put a + or – sign before a binary number to indicate its sign. For example $+1001_2$ indicates $+9_{10}$, and -1110_2 indicates -14_{10} etc.

- Signed number representations in Binary

When we store such a signed binary number in the computer, the computer uses special schemes to store the magnitude and the sign of that number. There are three basic methods of storing signed binary numbers. These are the **sign magnitude** representation, the **1's complement** representation and the **2's complement** representation. Each has its own merits and demerits. We now discuss below each of these methods of signed number representation.

Sign Magnitude

The Sign Magnitude Representation:

In this method, the **sign of a positive number is taken as '0'** and the **sign of a negative number is taken as '1'**. The MSB will be used for storing the sign of the number, while the remaining bits are used for storing the magnitude of the number.

Thus for an 8 bit representation of a binary number, the first bit from the left (i.e. the Most Significant Bit or MSB) will store the sign and the remaining 7 bits will store the magnitude. While for a 16 bit representation, the MSB will store the sign and the remaining 15 bits will store the magnitude.

In Sign Magnitude Representation a number is represented by a sign (0 for +, 1 for -) and its binary magnitude.

Example-82: Express +23 as an **8 bit sign magnitude number**.

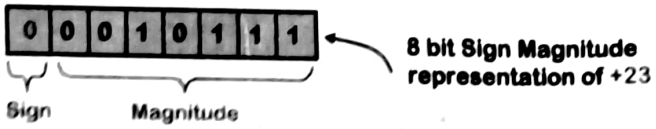
Step1: First convert the number to binary as shown below:

```
2 | 23
2 | 11   rem 1
2 |  5   rem 1
2 |  2   rem 1
2 |  1   rem 0
    0    rem 1
```

Therefore $23_{10} = (10111)_2$

Step2: Next write the magnitude of the number as a 7 bit number by **putting as many 0's to the left of the number as required** to make a 7 bit number. Therefore the number now becomes 0010111 with two leading zeroes.

Step3: Finally write the number along with the sign bit as an 8 bit number. Since the above number is positive, therefore the sign bit will be 0. The final number represented in 8 bit sign magnitude form is:

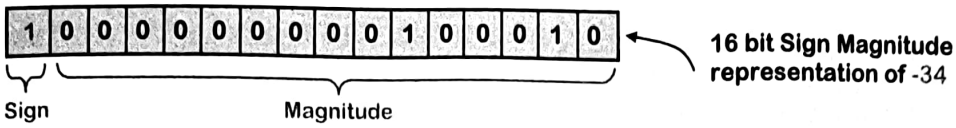


Example-83: Express –34 as a **16 bit sign magnitude number**.

2	34	
2	17	rem **0**
2	8	rem **1**
2	4	rem **0**
2	2	rem **0**
2	1	rem **0**
	0	rem **1**

Therefore $34_{10} = (100010)_2$

Since the number is negative, the sign of the number will be '1'. The number written as a 16 bit number along with the sign bit is shown below:



The 1's Complement Representation:



1's Complement Representation

In this method, for both positive and negative numbers, the number is **first expressed as a positive sign magnitude number** with the MSB representing the sign as '0'. If the original number is positive, then this positive sign magnitude number is the required answer.

However, if the original number is negative, then after finding the positive sign magnitude value, the **1's complement of that number** gives the required representation. To form the 1's complement, convert each 1 to 0 and each 0 to 1 in the positive sign magnitude number. The following example illustrates the process.

In 1's complement representation, the sign bit is 1 for negative numbers, and the magnitude is in complement form.

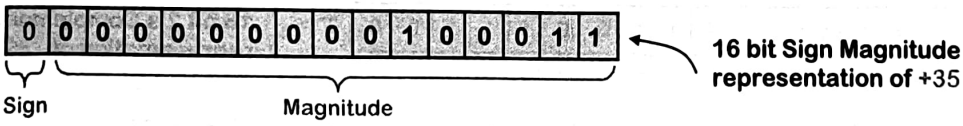
Example-84: Express –35 as a **16 bit 1's complement number**.

First convert the number to binary.

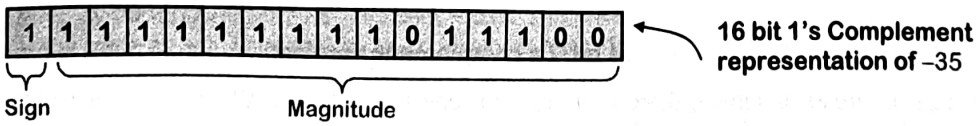
2	35	
2	17	rem **1**
2	8	rem **1**
2	4	rem **0**
2	2	rem **0**
2	1	rem **0**
	0	rem **1**

Therefore $35_{10} = (100011)_2$

The number is then **written as a 16 bit positive sign magnitude number** along with the sign bit as:

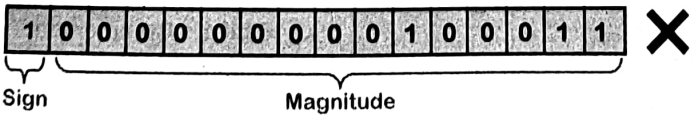


Finally the number is converted to 1's complement form by **replacing each 0 by 1 and each 1 by 0**.



Thus $(1111111111011100)_2$ is the required 16 bit 1's complement representation of -35_{10}.

Note that for a negative number the **sign bit of the final 1's complement will always be 1**. It is a **common mistake** to initially put 1 for the sign bit, when considering the sign magnitude representation of the number in the first step before converting it to 1's complement form. For example in the above example **it is wrong to write** the number with 1 in the sign bit in the first step as:



2's complement

The 2's Complement Representation:

In this method, for both positive and negative numbers, the number is **first expressed as a positive sign magnitude number** with the MSB representing the sign as '0'. If the original number is positive, then the positive sign magnitude number is also the required representation for the 2's complement.

However, if the original number is negative, then after finding the positive sign magnitude value, the **1's complement of that number** is found out. Then **1 is added to the 1's complement** to get the **2's complement representation**. The following example illustrates the process.

In an n-bit, signed, two's complement binary system, the largest number that can be formed is a 0 followed by all 1s, and the smallest is a 1 followed by all 0's.

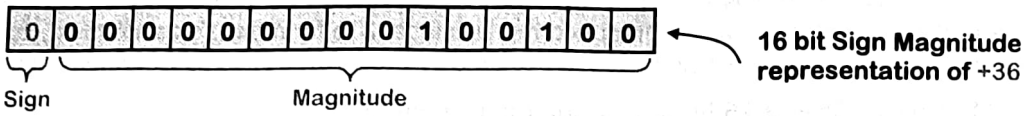
Example-85: Express –36 as a **16 bit 2's complement number**.

First convert the number to binary.

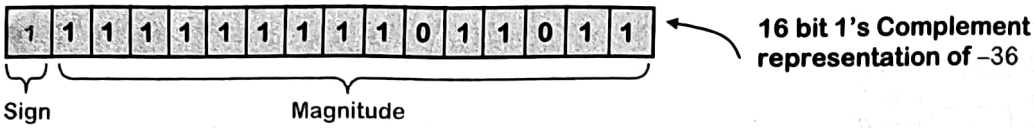
2	36	
2	18	rem **0**
2	9	rem **0**
2	4	rem **1**
2	2	rem **0**
2	1	rem **0**
	0	rem **1**

Therefore $36_{10} = (100100)_2$

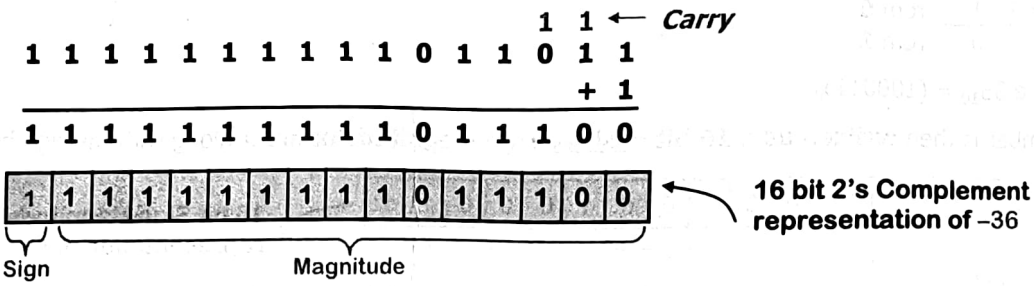
The number is then written as a 16 bit positive sign magnitude number along with the sign bit as:



Next the number is converted to 1's complement form by replacing each 0 by 1 and each 1 by 0.



Finally the number is converted to 2's complement form by adding 1 to the above number.



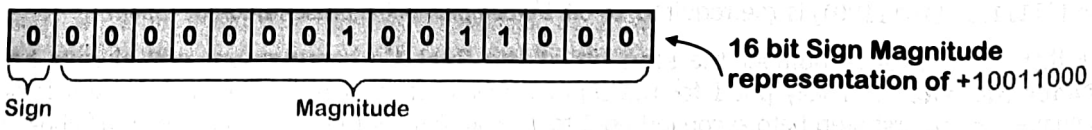
∴

Thus $(1111111111011100)_2$ is the required 16 bit 2's complement representation of -36_{10}

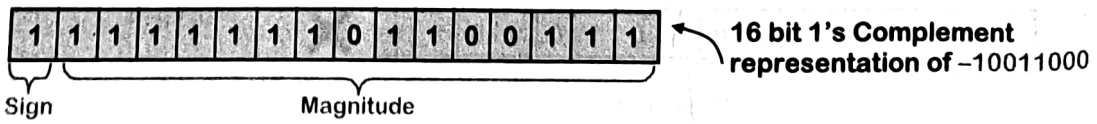
Note that for a negative number, the **sign bit of the 2's complement number is 1**.

Example-86: Express –10011000 as a **16 bit 2's complement number**.

As the number is already in binary, there is no need to convert it to binary. We first directly write the number as a 16 bit positive sign magnitude number as:

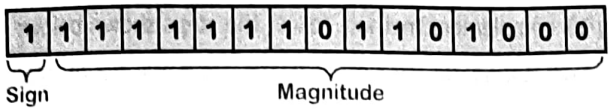


The 1's complement of the above number gives:



When 1 is added to the above number, we finally get the 2's complement of the number as shown.

```
                                  1 1 1 ← Carry
1 1 1 1 1 1 1 1 0 1 1 0 0 1 1 1
                          + 1
-------------------------------
1 1 1 1 1 1 1 1 0 1 1 0 1 0 0 0
```

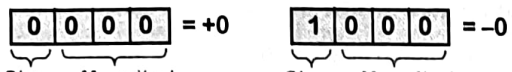
∴ 

16 bit 2's Complement representation of –10011000

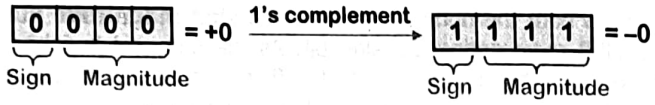
Thus $(1111111101101000)_2$ is the required 16 bit 2's complement representation of -10011000_2

- **Importance of 2's complement representation**

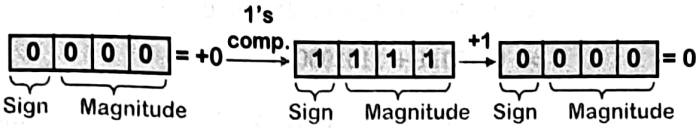
Of all the three signed number representations, the 2's complement form is the one that is used to represent negative binary numbers in the computer. To understand why, let us find the representations of '0' in various formats for a 4 bit binary number, with 3 bits for the magnitude and 1 bit for the sign.

In sign magnitude form, we have: 

We find that there are **two different representations for 0**, i.e. +0 and -0. However in reality we do not have a negative 0. Another disadvantage is that, out of a total of 2^4=16 possible combinations with 4 bits, 2 combinations are used to represent the same value i.e. 0.

In 1's complement form, we have: 

Again we find that there are **two different representations for 0**. If we convert all the 0's to 1's we get 1111, which represents the 1's complement form for -0. Therefore again we have 2 combinations used to represent the same value i.e. 0.

In 2's complement form, we have: 

To find the 2's complement representation of zero, we first found the 1's complement of 0000 to be 1111 and then added 1 to it to get the required representation as shown below:

```
  1 1 1 ← Carry
  1 1 1 1
      + 1
-------------
(1) 0 0 0 0
```

→ The **5th digit** (the final carry out) is **ignored** as the original number is 4 bits long

After adding 1 to the 1's complement form we get 10000. We **ignore the leftmost bit** (the 5th bit) which is equal to 1, as the original number had only 4 bits. This gives the final representation of -0 as also 0000, which is the same as +0. Therefore in 2's complement form **there is a single representation for 0**. There are no two different combinations that represent +0 and -0. Therefore **2's complement representation is a better representation** method for signed numbers than the sign magnitude method or the 1's complement method.

Importance of 2's complement

2's complement numbers have a single representation for zero. There is no +0 and -0.

Range of an 'n' bit number using the various signed number representations

System	Bits	Minimum Value	Maximum Value	Range
Sign Magnitude	n	$-[2^{(n-1)}-1]$	$+[2^{(n-1)}-1]$	$-[2^{(n-1)}-1]$ to $+[2^{(n-1)}-1]$
1's Complement	n	$-[2^{(n-1)}-1]$	$+[2^{(n-1)}-1]$	$-[2^{(n-1)}-1]$ to $+[2^{(n-1)}-1]$
2's Complement	n	$-2^{(n-1)}$	$+[2^{(n-1)}-1]$	$-2^{(n-1)}$ to $+[2^{(n-1)}-1]$

Range of an 'n' bit number

- **1's and 2's Complement Subtraction**

Unlike humans, it is more complex to do subtraction using a computer. Hence instead of using the method that humans use to do subtraction, the **computer uses an additive method** to do subtraction using the complement of the number to be subtracted as shown in this section.



1's complement subtraction

1's complement subtraction

To do subtraction of binary numbers using 1's complement method, follow the steps shown below:

Step1: Express both numbers as positive sign magnitude numbers with the same number of total bits

Step2: Put a sign bit as the leftmost bit of each number (MSB)

Step3: Find the **1's complement** of the number which is to be subtracted (i.e. the **subtrahend**)

Step4: **Add the complement** obtained in step1 to the number from which it is to be subtracted (i.e. the **minuend**)

Step5: In case there is an overflow carry generated at the end of the addition then **add the overflow carry with the result** obtained after the addition to obtain the final difference

Step6: In case the overflow carry is not present, then find the complement of the result obtained in step 2 and put a negative sign before it to get the final result (to indicate a negative result)

Example-87: Find $110011_2 - 10111_2$ using 1's complement method. Also carry out the decimal subtraction to check for the correctness of the result.

First express both the numbers as **positive sign magnitude numbers with the same number of bits** and the MSB representing the sign bit. As the minuend is 6 bits and the subtrahend is 5 bits, **both are expressed as 7 bit numbers** with the first bit representing the sign and the remaining 6 bits representing the magnitude. Therefore the above calculation is equal to:

In 1's complement subtraction, the overflow bit produced after addition is to be added to the LSB

$110011 - 10111 = 0110011 - 0010111$

$= 0110011 + 1\text{'s complement of } (0010111)$

$= 0110011 + 1101000$

Overflow bit

$$\begin{array}{r} {\scriptstyle 1\ \ \ \ \ \ \ } \\ 0\ 1\ 1\ 0\ 0\ 1\ 1_2 \\ +\ 1\ 1\ 0\ 1\ 0\ 0\ 0_2 \\ \hline (1)\ 0\ 0\ 1\ 1\ 0\ 1\ 1 \\ \qquad\qquad\quad +\ 1 \\ \hline 0\ 0\ 1\ 1\ 1\ 0\ 0 \end{array} \qquad \begin{array}{r} 51_{10} \\ -23_{10} \\ \hline 28_{10} \end{array}$$

$= 2^4x1 + 2^3x1 + 2^2x1 = 16 + 8 + 4 = 28_{10}$

In the above case, after the addition as there was an **overflow of 1**, it was **added to the result** of the addition to get the final result of the 1's complement subtraction.

The required **difference** is = $(11100)_2$

Example-88: Find $111011_2 - 101100_2$ using 1's complement method. Also carry out the decimal subtraction to check for the correctness of the result.

$111011 - 101100 = 0111011 - 0101100$

$= 0111011 + 1\text{'s complement of } (0101100)$

$= 0111011 + 1010011$

Overflow bit

$$\begin{array}{r} {\scriptstyle 1\ \ 1\ \ \ \ \ 1\ \ 1\ } \\ 0\ 1\ 1\ 1\ 0\ 1\ 1_2 \\ +\ 1\ 0\ 1\ 0\ 0\ 1\ 1_2 \\ \hline (1)\ 0\ 0\ 0\ 1\ 1\ 1\ 0 \\ \qquad\qquad\quad +\ 1 \\ \hline 0\ 0\ 0\ 1\ 1\ 1\ 1 \end{array} \qquad \begin{array}{r} 59_{10} \\ -44_{10} \\ \hline 15_{10} \end{array}$$

$= 2^3x1 + 2^2x1 + 2^1x1 + 2^0x1 = 8 + 4 + 2 + 1 = 15_{10}$

The required **difference** is = $(1111)_2$

Example-89: Find $1000010_2 - 1111000_2$ using 1's complement method. Also carry out the decimal subtraction to check for the correctness of the result.

In 1's complement subtraction, no overflow bit is produced after the addition if the result is negative.

$1000010 - 1111000 = 01000010 - 01111000$

$= 01000010 + 1\text{'s complement of } (01111000)$

$= 01000010 + 10000111$

$$\begin{array}{rr} & \overset{}{}\overset{1}{1}\overset{1}{} \\ & 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0_2 \\ + & 1\ 0\ 0\ 0\ 0\ 1\ 1\ 1_2 \\ \hline & 1\ 1\ 0\ 0\ 1\ 0\ 0\ 1 \end{array} \qquad \begin{array}{r} 66_{10} \\ -120_{10} \end{array} \Big\} \begin{array}{r} -120 \\ 66 \\ \hline -54_{10} \end{array}$$

NO Overflow bit, AND sign bit=1 ∴Negative answer

↓ 1's complement

$0\ 0\ 1\ 1\ 0\ 1\ 1\ 0$

Now $00110110_2 = (2^5x1 + 2^4x1 + 2^2x1 + 2^1x1) = (32+16+4+2) = 54_{10}$

In this example a larger number (120_{10}) is subtracted from a smaller number (66_{10}). The fact that the result should be negative is seen from the sign bit of the first addition. Note that unlike the previous subtraction **there is no overflow bit.** Also the **leftmost bit i.e. the sign bit is 1**. This indicates that the result is negative and hence **present in 1's complement form**. We will not get the actual magnitude from this 1's complement form. Hence the 1's complement operation is again done on the result to get the actual magnitude of the result. The final result is written by **placing a negative sign** before the value to get the result as –110110. **Do not forget** to put the **negative sign** in such case.

Therefore the required **difference** is $= -(110110)_2$

Example-90: Find $119_{10} - 126_{10}$ using 1's complement method.

First the numbers are converted to binary.

2 | 119		
2 | 59	rem 1	↑
2 | 29	rem 1	|
2 | 14	rem 1	|
2 | 7	rem 0	|
2 | 3	rem 1	|
2 | 1	rem 1	|
0	rem 1	|

$(119)_{10} = (1110111)_2$

2 | 126		
2 | 63	rem 0	↑
2 | 31	rem 1	|
2 | 15	rem 1	|
2 | 7	rem 1	|
2 | 3	rem 1	|
2 | 1	rem 1	|
0	rem 1	|

$(126)_{10} = (1111110)_2$

$\therefore 1110111 - 1111110 = 01110111 - 01111110$ [with the sign bit]

$= 01110111 + 1\text{'s complement of } (01111110)$

$= 01110111 + 10000001$

$$\begin{array}{rr} & \overset{}{}\overset{1}{1}\overset{1}{1}\overset{1}{} \\ & 0\ 1\ 1\ 1\ 0\ 1\ 1\ 1_2 \\ + & 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1_2 \\ \hline & 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0 \end{array} \qquad \begin{array}{r} 119_{10} \\ -126_{10} \end{array} \Big\} \begin{array}{r} -126 \\ 119 \\ \hline -7_{10} \end{array}$$

NO Overflow bit, AND sign bit=1 ∴Negative answer

↓ 1's complement

$0\ 0\ 0\ 0\ 0\ 1\ 1\ 1 = (2^2x1 + 2^1x1 + 2^0x1) = (4 + 2 + 1) = 7_{10}$

In the above example, since a larger number (126_{10}) is subtracted from a smaller number (119_{10}) the final result will be negative and is equal to -7_{10}. Thus we have to take the 1's complement of the first result of addition to get the final result and the answer is written with a negative sign.

Therefore the required **difference** is $= -(111)_2 = -7_{10}$

2's complement subtraction

2's complement subtraction

To subtract binary numbers using 2's complement method, the following steps need to be carried out:

Step1: Find the **2's complement** of the number which is to be subtracted (i.e. the **subtrahend**)

Step2: **Add the complement** obtained in step1 to the number from which it is to be subtracted (i.e. the **minuend**)

Step3: In case there is an overflow carry generated at the end of the addition then **ignore the overflow bit** to obtain the final difference

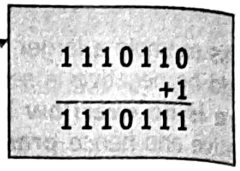
Step4: In case the overflow carry is not present, then **find the 2's complement of the result** obtained in step 2 and put a negative sign before it to get the final result

In 2's complement subtraction, the overflow bit produced after addition is to be neglected.

Example-91: Find $111011_2 - 1001_2$ using 2's complement method. Verify result using decimal subtraction.

First express both the numbers as **positive sign magnitude numbers with the same number of bits** and the MSB representing the sign bit. As the minuend is 6 bits and the subtrahend is 4 bits in this example, **both are expressed as 7 bit numbers** with the first bit representing the sign and the remaining 6 bits representing the magnitude. Therefore the above calculation is equal to:

111011 – 1001 = 0111011 – 0001001
= 0111011 + 2's complement of (0001001)
= 0111011 + [{ 1's complement of (0001001) } + 1]
= 0111011 + **[1110110+1]**
= 0111011 + 1110111



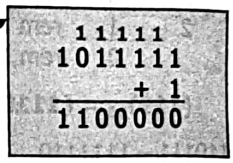
1 1 1 1 1 1
0 1 1 1 0 1 1$_2$ | 59_{10}
+ 1 1 1 0 1 1 1$_2$ | $-\ 9_{10}$
IGNORE Overflow bit → (1) 0 1 1 0 0 1 0 | 50_{10}

Also the result 01 1 0 0 1 0$_2$ = 2^5x1 + 2^4x1 + 2^1x1 = 32 + 16 + 2 = 50_{10}

In the above case, after the addition as there was an overflow of 1, **it was ignored** to get the final result of the 2's complement subtraction as $(110010)_2$

Example-92: Find $110011_2 - 100000_2$ using 1's complement method. Also carry out the decimal subtraction to check for the correctness of the result.

110011 – 100000 = 0110011 – 0100000
= 0110011 + 2's complement of (0100000)
= 0110011 + [{ 1's complement of (0100000) } + 1]
= 0110011 + **[1011111+1]**
= 0110011 + 1100000



1
0 1 1 0 0 1 1$_2$ | 51_{10}
IGNORE Overflow bit → + 1 1 0 0 0 0 0$_2$ | -32_{10}
(1) 0 0 1 0 0 1 1 | 19_{10}

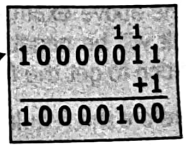
Also the result 00 1 0 0 1 1$_2$ = 2^4x1 + 2^1x1 + 2^0x1 = 16 + 2 + 1 = 19_{10}

Therefore the required **difference** is = $(10011)_2$

In 2's complement subtraction no overflow bit is produced after the addition if the result is negative.

Example-93: Find $1000011_2 - 1111100_2$ using 2's complement method. Carry out the decimal subtraction to check for the correctness of the result.

1000011 – 1111100 = **0**1000011 – **0**1111100
= 01000011 + 2's complement of (01111100)
= 01000011 + [{ 1's complement of (01111100) } + 1]
= 01000011 + **[10000011+1]**
= 01000011 + 10000100



0 1 0 0 0 0 1 1$_2$ | 67_{10} } −124
NO Overflow bit, AND sign bit=1 ∴Negative answer ← + 1 0 0 0 0 1 0 0$_2$ | -124_{10} } 67
1 1 0 0 0 1 1 1 | − 57_{10}

↓ 2's complement = 1's complement + 1

0 0 1 1 1 0 0 0$_2$ (1's complement)
+ 1
0 0 1 1 1 0 0 1 (gives the magnitude of the result)

Now $00111001_2 = (2^5x1+2^4x1+2^3x1+2^0x1) = (32+16+8+1) = 57_{10}$ (gives the magnitude here)

In this example a larger number (124_{10}) is subtracted from a smaller number (67_{10}). The fact that the result should be negative is seen from the sign bit of the first addition. Note that unlike the previous subtraction **there is no overflow bit**. Also the **leftmost bit i.e. the sign bit is 1**. This indicates that the result is negative and hence **present in 2's complement form**. We will not get the actual magnitude from this 2's complement form. Hence the 2's complement operation is again done on the result to get the actual magnitude of the result. The final result is written by **placing a negative sign** before the value to get the result as –111001. **Do not forget** to put the **negative sign** in such case.

Therefore the required **difference** is = $-(111001)_2$

Example-94: Find $21_{10} - 31_{10}$ using 2's complement method.

2 | 21		2 | 31	
2 | 10	rem **1**	2 | 15	rem **1**
2 | 5	rem **0**	2 | 7	rem **1**
2 | 2	rem **1**	2 | 3	rem **1**
2 | 1	rem **0**	2 | 1	rem **1**
0	rem **1**	0	rem **1**

$(21)_{10} = (10101)_2$ $(31)_{10} = (11111)_2$

$\therefore 10101 - 11111 = 010101 - 011111$ [with the sign bit]

$= 010101 +$ 2's complement of (011111)

$= 010101 + [\{$ 1's complement of $(011111)\} + 1]$

$= 010101 + [100000+1]$

$= 010101 + 100000$

100000 + 1 = 100001

$$\begin{array}{r} \overset{}{}\overset{1}{} \\ 0\ 1\ 0\ 1\ 0\ 1_2 \\ +\ 1\ 0\ 0\ 0\ 0\ 1_2 \\ \hline 1\ 1\ 0\ 1\ 1\ 0 \end{array} \quad \left| \begin{array}{r} 21_{10} \\ -\ 31_{10} \end{array} \right\} \begin{array}{r} -31 \\ 21 \\ \hline -\ 10_{10} \end{array}$$

NO Overflow bit, AND sign bit=1 ∴ Negative answer

2's complement = 1's complement + 1

$$\begin{array}{r} \overset{}{}\overset{1}{} \\ 0\ 0\ 1\ 0\ 0\ 1 \\ +\ \ 1 \\ \hline 0\ 0\ 1\ 0\ 1\ 0 \end{array}$$ (1's complement)

$= (2^3x1 + 2^1x1) = (8 + 2) = 10_{10}$ (gives the magnitude)

Therefore the required **difference** is = $-(1010)_2$

- **General Rule to find the Complement of a Number**

In general, the **radix or base complement** of an **n** digit number **x**, with base **b**, is by definition given as $\mathbf{b^n - x}$.

Thus for the decimal number system if the number is 2659, then the radix complement of 2659 will be $10^4 - 2659 = 10000 - 2659 = 7341$.

However, the radix complement is easily obtained by adding 1 to the **diminished radix complement**, which is given by $\mathbf{(b^n - 1) - x}$. The diminished radix complement can be found more easily by complementing each digit in the original number with respect to **(b-1)**. Thus for the decimal number 2659, the diminished radix complement i.e. 9's complement will be:

$(10^4 - 1) - 2659 = (10000 - 1) - 2659 = 9999 - 2659 = 7340$. By adding 1 to this we can easily get the same 10's complement as (7340+1) = 7341.

Similarly for the hexadecimal system with base as 16, the diminished radix complement of a number like 3A8D will be:

$(10^4 - 1) - 3A8D = (10000 - 1) - 3A8D = FFFF - 3A8D = C572_{16}$

Note that when 1 is subtracted from 10000, then the result will be FFFF in hexadecimal, as $FFFF+1=10000_{16}$ in hexadecimal number system. The radix complement of 3A8D can then be easily calculated as:

$C572_{16} + 1 = C573_{16}$.

General Rule for complements

3.6 Various Binary Coding Schemes

Unlike real numbers which have an infinite range, there are only a small finite number of characters like the alphabets, digits, punctuations etc. An entire character set can be represented with just a few bits per character. Some of the most common character representations are described below.

ASCII Code

- **The ASCII Code**

The **American Standard Code for Information Interchange** (**ASCII**) is used for representing characters in a computer system. The representation for each character consists of 7 bits, and all 2^7 possible bit patterns represent valid characters. A table for all the ASCII characters is given at the end of the chapter.

ASCII code *is used for representing the characters in a computer system. It uses 7 bits to represent a character.*

The ASCII character set (excluding the extended characters defined by IBM) is **divided into four groups of 32 characters**.

The **first group** of 32 ASCII characters from 0 through 31 (1F in Hex)) form a **special set of non-printing characters called the control characters**. They are called control characters because they perform various printer/display control operations rather than displaying symbols. Examples include *carriage return* which positions the cursor to the left side of the current line of characters, *line feed* which moves the cursor down one line on the output device, and back space which moves the cursor back one position to the left.

The **second group** of 32 ASCII characters comprise **various punctuation symbols special characters and the numeric digits**. The most notable characters in this group include the space character (ASCII code 32) and the numeric digits (ASCII codes 48..57). Note that by subtracting 48 from the ASCII code for any particular digit you can obtain the numeric equivalent of that digit.

The **third group** of 32 ASCII characters is **reserved for the upper case alphabetic characters**. The ASCII codes for the characters 'A'...'Z' lie in the range 65...90. Since there are only 26 different alphabetic characters the remaining six codes hold various special symbols.

The **fourth and final group** of 32 ASCII character codes are **reserved for the lower case alphabetic symbols, five additional special symbols**, and another control character (delete). Note that the lower case character symbols use the ASCII codes 97..122.

EBCDIC Code

- **The EBCDIC Code**

A problem with the ASCII code is that only 128 characters (from 0000000_2 to 1111111_2) can be represented, which is a limitation for many modern keyboards that have a host of special characters in addition to the standard characters. The **Extended Binary Coded Decimal Interchange Code** (**EBCDIC**) is an **eight-bit code** used widely in IBM mainframe computers and allows representing a maximum of 256 symbols.

ISCII Code

- **The Indian Script Code for Information Interchange (ISCII)**

Indian Standard Code for Information Interchange (**ISCII**) is a **coding scheme for representing various writing systems of India**. It encodes the main Indic scripts. The supported scripts are Assamese, Bengali, Devanagari, Gujarati, Gurmukhi, Kannada, Malayalam, Oriya, Tamil, and Telugu. It is an 8-bit encoding scheme. The lower 128 codes are plain ASCII, the upper 128 codes are ISCII-specific.

ISCII code *is used for representing the characters used in various writing systems in India.*

ISCII was proposed in the eighties and a suitable standard was evolved by 1991. Here are the **salient aspects of the ISCII representation**.

- It is a single representation for all the Indian Scripts.
- Codes have been assigned in the upper ASCII region (160-255) for the alphabets of the specific language.
- The scheme also assigns codes for the *matras*.
- Special characters have been included to specify how a consonant in a syllable should be rendered. Rendering of Devanagari has been kept in mind.
- A special attribute character has been included to identify the script to be used in rendering specific sections of the text.

	A0	B0	C0	D0	E0	F0
0		ओ	ढ	ॠ	०	EXT
1	ँ	औ	ण	ल	१	०
2	ं	ऑ	त	ळ	२	१
3	ः	क	थ	ऴ	३	२
4	अ	ख	द	व	४	३
5	आ	ग	ध	श	५	४
6	इ	घ	न	ष	६	५
7	ई	ङ	ऩ	स	७	६
8	उ	च	प	ह	८	७
9	ऊ	छ	फ	ॐ	९	८
A	ऋ	ज	ब	ऺ	॥	९
B	ऌ	झ	भ	ऻ		
C	ऍ	ञ	म	़		
D	ऎ	ट	य	ा		
E	ए	ठ	र	ि		
F	ऐ	ड	ऱ	ी	ATR	

In the **table shown in the last page**, there are six columns of 16 codes each, starting at the hexadecimal value of A0 which is equivalent to decimal 160. Note that some code values have not been assigned.

Note that **ISCII codes have nothing to do with fonts** and a given text in ISCII may be displayed using many different fonts for the same script. This will require specific rendering software which can map the ISCII codes to a matching font for the script.

- **The Binary Coded Decimal (BCD) Code**

Binary Coded Decimal

Since most circuits work using binary logic, hence the binary number system is the most suitable number system for computers. However, the human society is used to the decimal number system. Hence the values entered by humans in the decimal number system need to be converted to binary before calculations can be done by the computer. Mainly due to this reason, **most of the early computers worked using binary-coded decimal number systems**. In such a system, a coded group of binary bits are used to represent each of the ten decimal digits from 0 to 9.

BCD code is used to represent both digits and characters in computers and is a 6 bit code with 64 possible symbols.

Binary Coded Decimal (BCD) code is one such **alphanumeric code** developed by IBM and that was used to **represent both digits and characters in computers**. It is a **6-bit code** that can be used to represent a maximum of 2^6=64 possible symbols. Thus apart from the 10 digits from 0 to 9, only uppercase alphabets and a few other symbols can be represented by a BCD code.

In this code, each decimal digit is represented as a 4 bit binary number as shown on the right. Thus the digit 1 is represented as 0001, 5 as 0101, 9 as 1001 etc. It is also known as a **8421 code** as **each bit is assigned a weight depending upon its position**. Here the MSB has a weight of 8 and the LSB has a weight of 1.

Decimal	BCD code	Decimal	BCD code
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

To convert any BCD value to Decimal we have to just multiply the binary digits by their weights. Thus 1001_{BCD} = (1x**8** + 0x**4** + 0x**2** +1x**1**) = 9_{10}.

To represent a number like 309 using BCD, instead of finding the binary equivalent of 309, **each digit** is represented as its **corresponding BCD number** as shown below:

3 **0** **9** (in decimal)
0011 **0000** **1001** (in BCD code)

Therefore 309_{10} is (0011 0000 1001) in BCD code.

Similarly $(6485)_{10}$ = $(0110\ 0100\ 1000\ 0101)_{BCD}$ (refer to table for decimal digit to BCD conversion)

- **The Excess-3 Code:**

Excess-3 Code

Excess-3 code is a self complementing code i.e. the complement of a number in XS-3 directly gives the 9's complement value of that number.

A problem with the BCD code is the formation of complements. It is a common feature in a computer system to perform a subtraction by using the complement of the number to be subtracted. Thus [x – y] is calculated as [x + complement of y]. For example with decimal numbers the 9's complement is used for subtraction, whereby $[X_{10} - Y_{10}] = [X_{10} + (9\text{'s complement of } Y_{10})]$.

However when working with a computer, the most natural complement is the 1's complement, whereby the 0's and the 1's in the number are simply reversed to get the complement of the number. However, using the BCD code you do not get the complement directly or naturally. For example the 9's complement of 7_{10} is $(9\text{-}7)=2_{10}$. But since the BCD of 7 is 0111_2, the natural complement of 0111_2 is 1000_2, which does not represent the BCD value for 2 i.e. 0010_2.

One of the first codes developed to overcome this problem was Excess-3 code developed at Harvard and used with the Mark machines. In this system the **number 0011_2 (i.e. 3_{10}) is added to each BCD value to get its equivalent Excess-3 code.**

9's complement

Decimal	XS-3 code	Decimal	XS-3 code
0	0011	9	1100
1	0100	8	1011
2	0101	7	1010
3	0110	6	1001
4	0111	5	1000

1's complement

The table on the right gives the excess 3 codes for the decimal digits 0 to 9. Note that the code is a **self-complementing code**. This means that the complement of a number in XS-3 code directly gives the corresponding 9's complement value of that number.

The following examples convert the decimal numbers **469** and **530** to their XS-3 codes.

4	6	9	(Decimal)	5	3	0	(9's complement Decimal)
0100	0110	1001	(**BCD code**)	0101	0011	0000	
+0011	0011	0011	(*Addition of 3*)	+0011	0011	0011	
0111	1001	1100	(**XS-3 code**)	1000	0110	0011	(**XS-3 code for 530_{10}**)

9's complement

1's complement

One can observe that 530 is the 9's complement of 469 and at the same time the XS-3 code of 530 is the **direct 1's complement** of the XS-3 code for 469. Because of this, doing arithmetic calculations in XS-3 is easier than in BCD. However **XS-3 code is not a weighted code.**

Gray Code

- The Gray Code:

Gray code numbers are usually used instead of binary numbers to design analogue to digital counters. Consider the consecutive binary numbers 0111_2 (7) and 1000_2 (8). In a counter, when the counting changes from 7 to 8, all the four digits 0111 will get changed to 1000. This change in all the digits usually does not occur simultaneously. As a result **we can get intermediate numbers in-between the change, producing erroneous results.**

XOR Logic

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

In Gray code system, the bit change between any two consecutive numbers is always only one.

The Gray code was developed to overcome this problem. **In the Gray code system, the bit change between any two consecutive numbers is always only one.** A Gray code for a given number can be obtained by successive XOR operation between the bits using the following rule:

Decimal	Binary	Gray Code
0	000	**000**
1	001	**001**
2	010	**011**
3	011	**010**
4	100	**110**
5	101	**111**
6	110	**101**
7	111	**100**

If $b_3\ b_2\ b_1\ b_0$ is a binary number whose equivalent Gray code is $g_3\ g_2\ g_1\ g_0$, then we can obtain the values of the Gray code digits as:

$g_3 = b_3$ (No XOR operation is done on the MSB)
$g_2 = b_3 \oplus b_2$
$g_1 = b_2 \oplus b_1$
$g_0 = b_1 \oplus b_0$

MSB remains same

Thus binary $b_2\ b_1\ b_0$ = 111 = (1) (1⊕1) (1⊕1) = **100 in Gray code**
and binary $b_3\ b_2\ b_1\ b_0$ = 1010 = (1) (1⊕0) (0⊕1) (1⊕0) = **1111 in Gray code.**

Gray code is not a weighted code, as a digit in this system does not carry any positional value. The Gray code for a 3 bit counter that can count from 0 to 7 in binary is shown above. Note that though there are multiple changes in bit pattern between some of the consecutive numbers in the binary value column, but there is only a single bit change for any two consecutive values in the corresponding Gray code column. This **prevents the occurrence of any transient error values** when the count changes from one value to the next.

Bit map images

3.7 Bitmap Images

Apart from numeric and character data, a digital computer can also store image data in a digital form. Such a **digital image is basically a binary representation of a two-dimensional image.**

Depending upon the way the image is constructed, digital images **can be stored either as a raster image** or as a **vector image.** By default, a digital image usually refers to a raster image only.

Raster images are composed of picture elements called pixels, which form a bitmap image.

a. **Raster images are composed of a set of digital values called picture elements or pixels.** Pixels form the smallest building block of a digital image that can be accessed and processed. Each pixel stores information regarding the intensity of a given colour at any specific point in the image. The whole image is stored as a two dimensional array, consisting of rows and columns of these pixels. Such a method of **storing a colour picture in a digital computer is known as bitmap representation of an image.** In MS Windows, the image created by the application *Paintbrush* is a bitmap image that is saved with the **.bmp** extension.

Bitmap images can be classified based on the colour values of these pixels as:

- **Binary images**: Images consisting of only two colours (like only black and white)
- **Grayscale images**: Images consisting of shades of gray from pure white to pure black
- **Colour images**: Images consisting of multi-colour information

b. **Vector images are generated by mathematical equations**. For example, instead of storing a circle as a set of pixels, it can be stored as an equation like $ax^2+by^2+c=0$, which requires much less space to store the same data. However, these are not suitable for storing photographs.

Vector images are generated by mathematical equations.

Originally bitmaps used one bit per pixel, without any colour information. But nowadays a multi-coloured image can also be stored as a bitmap, which is also called pixmap. **Various bitmap file formats** are in use. The common among these are the standardised compressed bitmap files such as GIF, JPEG, TIFF, and PNG.

3.8 Concept of Fixed and Floating Point Numbers

Fixed and Floating point Numbers

We have seen that computers can store only binary numbers and have learnt how to represent numbers in various number systems. In this section we will learn about the methods used to represent real numbers, i.e. numbers that can have a whole number part and a fractional part. Examples include numbers like 123.456, 0.0045, –67.0988, –0.0034506, 1.2898×10^{20} etc.

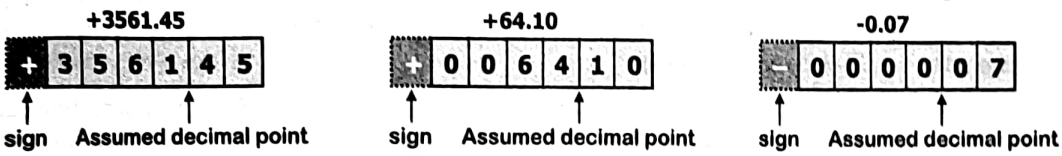
For storing such numbers we have to basically store the sign of the number, the digits that make the number and the position of the base point in the number. Depending upon the way these three things are stored, there are different types of representations of real numbers. Computers use specific amount of space for storing a given value. Various methods have been devised to store various numbers using the fixed space available. These are discussed below.

- Fixed Point Decimal Numbers

When the decimal point is at a fixed position within the set of available places for the digits, it is called a Fixed Point number.

The idea of fixed-point representation is to **put the decimal point at a fixed position within the set of available places.** This provides a fixed set of places to the left and right of the decimal point.

Let us assume that we are working with computer memory that can store a total of 6 decimal digits, with 4 digits storing the integer part and 2 digits storing the fraction part. The sign of the number is stored as an additional space to the left of the number. The examples below show how the numbers +3561.45, +64.1, and -0.07 are stored in this representation. The decimal point is assumed to be 2 places from the right.



Using the fixed representation as shown above, we can store a maximum positive value of +9999.99 and a maximum negative value of -9999.99. Thus the range of values that can be represented using this 6 position fixed point representation is from -9999.99 to +9999.99 only.

- Disadvantages of Fixed Point Numbers

Fixed point real numbers can be used in situations **where very large numbers are not involved**, or where **the precision of a value is not required** beyond a certain range. For example in financial accounting software, there is no need to store values with more than two places after the decimal point.

However, this method **cannot represent a wide range of values and may need a large number of digits to do so**. For example a system that can store both the values 0.0000000098765 and 126450000000.0 with correct precision would require a **lot of hardware to store and manipulate such numbers**. Also more is the number of digits, more is the time taken for the computation.

In this context note the representation of decimal numbers using a total of 6 digits as shown above. Though space is available for storing 6 decimal digits, we cannot store numbers like 123854 or 0.07056, though each of these numbers is actually requiring a maximum of six digits.

Fine precision however is generally not needed when very large or very small numbers are used. For example for majority of practical use it is sufficient to express the speed of light as 299792458 m/sec. The value is so large that whether it is 299792458.16 m/sec or 299792457.88 m/sec hardly makes a difference. In an efficient computer only **as much precision is retained as is required**.



Scientific Notation

• Scientific Notation Representation of Real Numbers

The second method, i.e. the **floating point representation of real numbers** is **based on the scientific notation** for expressing both very large and small numbers in a concise format. It **increases the range** of numbers that can be represented using a fixed set of positions. The representation allows a large range of numbers to be represented using a relatively smaller number of digits.

For example consider the decimal number 6023000000000000000000000. Though a very large value, but **can be represented using a concise notation** with the real number part 6.023, and the base 10 (for decimal numbers) raised to the power 23 as 6.023×10^{23}.

This **scientific notation** has therefore three components.

- **Base**: Indicating the base of the number system used (i.e. 10 for the decimal number above)
- **Exponent**: Indicting the power to which the base is raised (i.e. 23 for the number 6.023×10^{23})
- **Mantissa**: Indicating number of significant digits used to represent the number (i.e. 6.023 for the number 6.023×10^{23})

Using this form however the **same number can be represented in different ways**, which makes comparisons and arithmetic operations difficult. For example consider the value $(3584.1)_{10}$

$$
\begin{aligned}
(384.1)_{10} &= 384.1 \times 10^{0} \\
&= 38.41 \times 10^{1} \\
&= 3.841 \times 10^{2} \\
&= 0.3841 \times 10^{3} \\
&= 0.03841 \times 10^{4} \text{ etc.}
\end{aligned}
$$



Normalisation of Floating Point numbers

• Normalisation of Floating Point Numbers:

In a real number, the digits that are used to express the number and contribute to its precision are called the **significant digits**. For example 23.204_{10} and 101.11_{2} have 5 significant digits each, as all 5 digits are required to know the value of the numbers correctly. However -0.00056_{10} has only 2 significant digits i.e. 5 and 6, while the remaining three '0's after the decimal point are used for determining the position of the decimal point in the number, and hence are not considered as significant digits. Significant digits play an important role in representing numbers in floating point representation (note that for a number like 9.05600, the number of significant digits will be 6 and not 4, as by keeping the last two 0's we get an idea of the precise value up to the 5th place after decimal).

A normalized number is formed by shifting the base point to the right or left until it is to the left of the first significant digit. The exponent is adjusted accordingly

In order to avoid multiple representations for the same number as shown above **floating point numbers are stored in a pre-defined normalised form**. The process of normalisation involves **shifting the base point to the left or right** until the base point is to the left of the leftmost nonzero digit. To keep the value of the number unchanged, the exponent is adjusted accordingly.

In other words, **in a normalised number the base point is placed to the left of the first non-zero digit** and the exponent adjusted accordingly to keep the value of the number unchanged. The normalised values for some numbers are given below (*):

$$
\begin{aligned}
+(459.009)_{10} &= +0.459009 \times 10^{3} \\
+(987)_{10} &= +0.978 \times 10^{3} \\
-(0.987)_{10} &= -0.978 \times 10^{0} \\
+(0.000978)_{10} &= +0.978 \times 10^{-3} \\
-(0.001001)_{2} &= -0.1001 \times 2^{-2}
\end{aligned}
$$

The **general format** for representing a normalised floating point number N is: $\mathbf{N = s\ M \times R^{E}}$ where:

- **s** is the **sign** of the number i.e. + or – (the + sign may be omitted)
- **M** is called the **mantissa**, and forms the string of significant digits of the normalised floating point number. The value of the mantissa is between 0 and 1, i.e. $0<M<1$
- **R** is the **radix** or base of the number system used to represent the number
- **E** is the **exponent** part, which determines the position of the base point in the number

Hence if $N = -0.56 \times 10^{-2}$, **s** is equal to '–', **M** is equal to 0.56, **R** is equal to 10, and **E** is equal to -2.

(*) *Some systems represent a normalised number by putting the base point after the first significant digit as $+3.059 \times 10^{4}$ for example*

- **IEEE 754 binary floating point number standard:**

IEEE 754 Standard

IEEE Standard 754 floating point number representation is the **most common representation today for real numbers on computers**, including Intel-based PC's, Macintoshes, and most UNIX platforms. It was developed way back in the 80's by the Institution of Electrical and Electronics Engineers (IEEE), USA for representing floating point numbers in computers and performing arithmetic operations.

As in the previous case, IEEE floating point numbers have three basic components. These are the **sign, the exponent, and the mantissa part**. There are generally two types of numbers that are represented using this standard. These are the **32 bit single precision number and the 64 bit double precision** number. The distribution of bits for each representation is given below:

Type	Total	Sign	Exponent	Mantissa
Single Precision	32 bits	1 bit	8 bits	23 bits
Double Precision	64 bits	1 bit	11 bits	52 bits

A detailed discussion on this standard is beyond the scope of the syllabus.

- **Loss in precision in floating point representation:**

The error produced by truncating the least significant digits of a number is called Round-Off error

For a single precision number we are **using a 32-bit format and re-arranging the fields to cover a much broader range**. In doing so, we are actually **losing in precision**. For example, a regular 32-bit integer can precisely store any value from 0 to 2^{32}-1 = 4,294,967,295. A single-precision floating-point number on the other hand is unable to match this resolution with its 23 bits that are allotted for storing its significant digits. In this context note that a 23 bit binary integer can store values up to 2^{23}-1 = 8,388,607 which is much less than 4,294,967,295. So a 32 bit floating point value, with only 23 bits available for storing the significant digits **approximates this value by effectively truncating digits from the lower end** i.e. the LSB positions, with minimum loss in significance.

Though the range of floating point numbers is much more than integer numbers represented by using the same number of bits, **the price we have to pay for this higher range is a loss in precision**. The error produced by truncating the least significant digits of a number is called **round-off error**.

Round Off error

The ability for a range/precision compromise is a major advantage of using a floating point representation. In applications requiring higher precision one can increase the mantissa size, and in applications requiring a larger range, one can increase the exponent size, but cannot do both simultaneously.

- **Floating Point Arithmetic**

Floating point Arithmetic

This section deals with arithmetic operations like addition, subtraction, multiplication, and division that can be carried out on normalised floating point numbers.

Floating point Addition and Subtraction:

If two numbers in floating point representation are to be added or subtracted, the **exponents of the two numbers must be made equal first**. This may require shifting the radix point. To preserve the range of the numbers correctly, the **radix point needs to be shifted in the number with the lesser exponent**, to match the exponent of the number with the larger exponent.

Example-95: Add 0.2451×10^{7} and 0.5353×10^{7}

As the exponents are same for both the numbers we need to only add the mantissa portions as:
$0.2451 \times 10^{+7} + 0.5353 \times 10^{+7} = (0.2451 + 0.5353) \times 10^{7} = \mathbf{0.7804 \times 10^{+7}}$

Example-96: Add 0.1234 *E*03 and 0.4567 *E*02 (here *E*03 means $x10^{3}$, and *E*02 means $x10^{2}$)

Here the exponents are not the same for the numbers. Hence we need to first adjust the decimal point of the number with the lower exponent to match that with the higher exponent before adding the corresponding mantissa parts.

$0.1234 \times 10^{+3} + 0.4567 \times 10^{+2} = 0.1234 \times 10^{3} + 0.4567 \times 10^{3-1}$
$= 0.1234 \times 10^{3} + 0.4567 \times 10^{-1} \times 10^{3}$
$= 0.1234 \times 10^{3} + 0.0456 \times 10^{3}$
$= (0.1234 + 0.0456) \times 10^{3}$
$= \mathbf{0.1690 \times 10^{+3}}$

Note that after adjusting the exponent of the second number, we have not retained the last digit 7. The reason is that in a fixed space representation the number of digits available in the register is fixed. Accordingly we can store only the same number of digits as there are initially.

Example-97: Add 0.6321 *E*08 and 0.5736 *E*08

Here the exponents are same. Hence we simply need to add the mantissa parts.

$0.6321 \times 10^{+8} + 0.5736 \times 10^{+8} = (0.6321 + 0.5736) \times 10^{8}$
$= 1.2057 \times 10^{8}$
$= 0.12057 \times 10^{9}$ (after normalizing the result)
$= \mathbf{0.1205 \times 10^{+9}}$ (after removing the extra digit)

Note that the last digit in the result i.e. 7, has been truncated as there are only 4 places available in the register for storing the number.

Example-98: Add 0.5567×10^{99} and 0.7689×10^{99}

Here the exponents are same. Hence we simply need to add the mantissa parts.

$0.5567 \times 10^{+99} + 0.7689 \times 10^{+99} = (0.5567 + 0.7689) \times 10^{99}$
$= 1.3256 \times 10^{99}$
$= 0.13256 \times 10^{+100}$

Note that after normalising the result we have got an exponent of +100, which is beyond the storage capacity of the format used. Hence this **gives rise to an overflow condition**.

Example-99: Add 0.3876×10^{-8} and 0.9563×10^{-6}

Here the exponents are not the same for the numbers. Hence we need to first adjust the decimal point of the number with the lower exponent to match that with the higher exponent before adding the corresponding mantissa parts. Since the exponents are negative, we have -6 > -8 and hence both the exponents are made equal to the power 10^{-6} and the decimal point adjusted accordingly.

$0.3876 \times 10^{-8} + 0.9563 \times 10^{-6} = 0.3876 \times 10^{-6-2} + 0.9563 \times 10^{-6}$
$= 0.3876 \times 10^{-2} \times 10^{-6} + 0.9563 \times 10^{-6}$
$= 0.003876 \times 10^{-6} + 0.9563 \times 10^{-6}$
$= 0.0038 \times 10^{-6} + 0.9563 \times 10^{-6}$ (truncating the extra digits)
$= (0.0038 + 0.9563) \times 10^{-6}$
$= \mathbf{0.9601 \times 10^{-6}}$

Example-100: Add 0.1001×2^{8} and 0.0111×2^{7}

Here the numbers are expressed in binary and the exponents are not the same for the numbers. Hence we need to first adjust the binary point of the number with the lower exponent to match that with the higher exponent before adding the corresponding mantissa parts.

$0.1001 \times 2^{8} + 0.0111 \times 2^{7} = 0.1001 \times 2^{8} + 0.0111 \times 2^{8-1}$
$= 0.1001 \times 2^{8} + 0.0111 \times 2^{-1} \times 2^{8}$
$= 0.1001 \times 2^{8} + 0.00111 \times 2^{8}$
$= 0.1001 \times 2^{8} + 0.0011 \times 2^{8}$ (truncating the extra digit)
$= (0.1001 + 0.0011) \times 2^{8}$
$= \mathbf{0.1100 \times 2^{8}}$

Example-101: Add $0.1001_2 \times 2^{8}$ and $0.0111_2 \times 2^{7}$

Here the numbers are expressed in binary and the exponents are not the same for the numbers. Hence we need to first adjust the binary point of the number with the lower exponent, to match that with the higher exponent before adding the corresponding mantissa parts.

$0.1001 \times 2^{8} + 0.0111 \times 2^{7} = 0.1001 \times 2^{8} + 0.0111 \times 2^{8-1}$
$= 0.1001 \times 2^{8} + 0.0111 \times 2^{-1} \times 2^{8}$
$= 0.1001 \times 2^{8} + 0.00111 \times 2^{8}$
$= 0.1001 \times 2^{8} + 0.0011 \times 2^{8}$ (truncating the extra digit)
$= (0.1001 + 0.0011) \times 2^{8} = \mathbf{0.1100 \times 2^{8}}$

Example-102: Evaluate the expression $11.101 \times 2^9 + 1.0111 \times 2^8$ using floating point arithmetic.

Here the numbers are not in a normalised form and the exponents are also not the same. Hence we need to first normalise the numbers and then adjust the binary point of the number with the lower exponent, to match that with the higher exponent before adding the corresponding mantissa parts.

$11.101 \times 2^9 + 1.0111 \times 2^8 = 0.11101 \times 2^{11} + 0.10111 \times 2^9$ (normalising the numbers)

$= 0.11101 \times 2^{11} + 0.10111 \times 2^{11-2}$

$= 0.11101 \times 2^{11} + 0.0010111 \times 2^{11}$

$= 0.11101 \times 2^{11} + 0.00101 \times 2^{11}$ (truncating the extra digits)

$= (0.11101 + 0.00101) \times 2^{11}$

$= 1.00010 \times 2^{11}$

$= 0.100010 \times 2^{12}$ (normalising the result)

$= \mathbf{0.10001 \times 2^{12}}$ (removing the last extra digit)

Example-103: Perform the subtraction $0.6321\ E11 - 0.5736\ E11$ using floating point arithmetic.

Here the exponents are same. Hence we simply need to subtract the mantissa parts.

$0.6321 \times 10^{+11} - 0.5736 \times 10^{+11} = (0.6321 - 0.5736) \times 10^{11}$

$= 0.0585 \times 10^{11}$

$= 0.0585 \times 10 \times 10^{10}$

$= \mathbf{0.5850 \times 10^{10}}$ (after normalising the result)

Example-104: Evaluate $0.11010111 \times 2^5 - 0.10101011 \times 2^4$ using floating point arithmetic.

Here the exponents are not same and need adjusting the binary point before subtraction.

$0.11010111 \times 2^5 - 0.10101011 \times 2^4 = 0.11010111 \times 2^5 - 0.010101011 \times 2^5$

$= 0.11010111 \times 2^5 - 0.01010101 \times 2^5$ (removing the extra digit)

$= (0.11010111 - 0.01010101) \times 2^5$

$= \mathbf{0.10000010 \times 2^5}$

Example-105: Perform the subtraction $(0.63652E\text{-}12) - (0.7654E\text{-}11)$ using floating point arithmetic.

$0.63652 \times 10^{-12} - 0.7654 \times 10^{-11} = 0.63652 \times 10^{-11-1} - 0.76540 \times 10^{-11}$ (as -11 > -12)

$= 0.63652 \times 10^{-1} \times 10^{-11} - 0.76540 \times 10^{-11}$

$= 0.063652 \times 10^{-11} - 0.76540 \times 10^{-11}$

$= 0.06365 \times 10^{-11} - 0.76540 \times 10^{-11}$ (removing the extra digit)

$= (0.06365 - 0.76540) \times 10^{-11} = \mathbf{-0.70175 \times 10^{-11}}$

Example-106: Perform the subtraction $0.8301 \times 10^3 - 0.301 \times 10^2$ using floating point arithmetic.

$0.8301 \times 10^3 - 0.301 \times 10^2 = 0.8301 \times 10^3 - 0.0301 \times 10^3$

$= (0.8301 - 0.0301) \times 10^3$

$= \mathbf{0.8 \times 10^3}$



Floating point Multiplication and Division

Floating point Multiplication and Division:

To multiply two numbers in normalised floating point representation, first **multiply the corresponding mantissa** of both the numbers to get the mantissa of the result. Next **add the exponents** of each of the numbers to get the exponent of the result.

To divide two numbers in normalised floating point representation, first **divide the mantissa of the first number with the mantissa of the second number** to get the mantissa of the result. Next **subtract the exponent of the second number from the exponent of the first** to get the exponent of the result.

Example-107: Multiply 0.1234×10^{03} and 0.7689×10^{05} using floating point arithmetic.

$(0.1234 \times 10^3) \times (0.7689 \times 10^5) = (0.1234 \times 0.7689) \times 10^{3+5}$

$= 0.09488226 \times 10^8$

$= 0.9488226 \times 10^7$ (normalising the result)

$= \mathbf{0.9488 \times 10^7}$ (removing the extra digits from the end)

Example-108: Multiply 0.11010111×2^{4} and 0.10010011×2^{-2} using floating point arithmetic.

$(0.11010111 \times 2^{4}) \times (0.10010011 \times 2^{-2}) = (0.11010111 \times 0.10010011) \times 2^{4-2}$

$= 0.0111101101110101 \times 2^{2}$

$= 0.111101101110101 \times 2^{1}$ (normalising the result)

$= \mathbf{0.11110110 \times 2^{1}}$ (removing the extra digits)

Example-109: Multiply 0.9876×10^{47} and 0.6548×10^{56} using floating point arithmetic.

$(0.9876 \times 10^{47}) \times (0.6548 \times 10^{56}) = (0.9876 \times 0.6548) \times 10^{47+56}$

$= 0.64668048 \times 10^{103}$

$= 0.6466 \times 10^{103}$ (removing the extra digits)

Note that with a 2 digit exponent value the exponent 103 will **produce an overflow condition**.

Example-110: Multiply 0.2345×10^{-3} and 0.2679×10^{-2} using floating point arithmetic.

$(0.2345 \times 10^{-3}) \times (0.2679 \times 10^{-2}) = (0.11010111 \times 0.10010011) \times 2^{-3-2}$

$= 0.06282255 \times 2^{-5}$

$= 0.6282255 \times 2^{-6}$ (normalising the result)

$= \mathbf{0.6282 \times 2^{-6}}$ (removing the extra digits)

Example-111: Divide 0.1234×10^{05} by 0.7689×10^{03} using floating point arithmetic.

$(0.1234 \times 10^{05}) \div (0.7689 \times 10^{03}) = (0.1234 \div 0.7689) \times 10^{5-3}$

$= \mathbf{0.1604 \times 10^{2}}$ (after removing the extra digits)

Example-112: Divide 0.8634×10^{34} by 0.1197×10^{-69} using floating point arithmetic.

$(0.8634 \times 10^{34}) \div (0.1197 \times 10^{-69}) = (0.8634 \div 0.1197) \times 10^{34-(-69)}$

$= 7.2130 \times 10^{103}$

$= 0.72130 \times 10^{104}$ (normalising the result)

Note that with a 2 digit exponent value the exponent 104 will produce an **overflow condition**.

Example-113: Divide 0.9753×10^{-18} by 0.1364×10^{-20} using floating point arithmetic.

$0.9753 \times 10^{-18} \div 0.1364 \times 10^{-20} = (0.9753 \div 0.1364) \times 10^{-18-20}$

$= 7.15029 \times 10^{-38}$

$= 0.715029 \times 10^{-37}$ (normalising the result)

$= \mathbf{0.7150 \times 10^{-37}}$ (removing the extra digits)

Example-114: Divide 0.1101×2^{5} by 0.1×2^{3} using floating point arithmetic.

$0.11010 \times 2^{5} \div 0.1 \times 2^{3} = 0.1101 \times 2^{5} \div 0.1000 \times 2^{3}$

$= (0.1101 \div 0.1000) \times 2^{5-3}$

$= 1.101 \times 2^{2}$

$= \mathbf{0.1101 \times 2^{3}}$ (normalising the result)

Example-115: Divide 0.11011101×2^{8} by 0.11×2^{2} using floating point arithmetic.

$0.11011101 \times 2^{8} \div 0.11 \times 2^{2} = 0.11011101 \times 2^{8} \div 0.1100000 \times 2^{2}$

$= (0.11011101 \div 0.1100000) \times 2^{8-2}$

$= 1.00100110 \times 2^{6}$

$= 0.100100110 \times 2^{7}$ (normalising the result)

$= \mathbf{0.10010011 \times 2^{7}}$ (removing the extra digit)

The ASCII code is summarised below:

Dec.	Binary	Value	Remarks
000	00000000	NUL	(Null character)
001	00000001	SOH	(Start of Header)
002	00000010	STX	(Start of Text)
003	00000011	ETX	(End of Text)
004	00000100	EOT	(End of Transmission)
005	00000101	ENQ	(Enquiry)
006	00000110	ACK	(Acknowledgment)
007	00000111	BEL	(Bell)
008	00001000	BS	(Backspace)
009	00001001	HT	(Horizontal Tab)
010	00001010	LF	(Line Feed)
011	00001011	VT	(Vertical Tab)
012	00001100	FF	(Form Feed)
013	00001101	CR	(Carriage Return)
014	00001110	SO	(Shift Out)
015	00001111	SI	(Shift In)
016	00010000	DLE	(Data Link Escape)
017	00010001	DC1	(XON) (Device Control 1)
018	00010010	DC2	(Device Control 2)
019	00010011	DC3	(XOFF)(Device Control 3)
020	00010100	DC4	(Device Control 4)
021	00010101	NAK	(Negative Ack.)
022	00010110	SYN	(Synchronous Idle)
023	00010111	ETB	(End of Trans. Block)
024	00011000	CAN	(Cancel)
025	00011001	EM	(End of Medium)
026	00011010	SUB	(Substitute)
027	00011011	ESC	(Escape)
028	00011100	FS	(File Separator)
029	00011101	GS	(Group Separator)
030	00011110	RS	(Request to Send)
031	00011111	US	(Unit Separator)
032	00100000	SP	(Space)
033	00100001	!	(exclamation mark)
034	00100010	"	(double quote)
035	00100011	#	(number sign)
036	00100100	$	(dollar sign)
037	00100101	%	(percent)
038	00100110	&	(ampersand)
039	00100111	'	(single quote)
040	00101000	(	(left parenthesis)
041	00101001	)	(right parenthesis)
042	00101010	*	(asterisk)
043	00101011	+	(plus)
044	00101100	,	(comma)
045	00101101	-	(minus or dash)
046	00101110	.	(dot)
047	00101111	/	(forward slash)
048	00110000	0	Start of the 10 dec. digits
049	00110001	1	
050	00110010	2	
051	00110011	3	
052	00110100	4	
053	00110101	5	
054	00110110	6	
055	00110111	7	
056	00111000	8	
057	00111001	9	
058	00111010	:	(colon)
059	00111011	;	(semi-colon)
060	00111100	<	(less than)
061	00111101	=	(equal sign)
062	00111110	>	(greater than)
063	00111111	?	(question mark)

Dec.	Binary	Value	Remarks
064	01000000	@	(AT symbol)
065	01000001	A	Start of Upper Case
066	01000010	B	
067	01000011	C	
068	01000100	D	
069	01000101	E	
070	01000110	F	
071	01000111	G	
072	01001000	H	
073	01001001	I	
074	01001010	J	
075	01001011	K	
076	01001100	L	
077	01001101	M	
078	01001110	N	
079	01001111	O	
080	01010000	P	
081	01010001	Q	
082	01010010	R	
083	01010011	S	
084	01010100	T	
085	01010101	U	
086	01010110	V	
087	01010111	W	
088	01011000	X	
089	01011001	Y	
090	01011010	Z	
091	01011011	[	(left/opening bracket)
092	01011100	\	(back slash)
093	01011101	]	(right/closing bracket)
094	01011110	^	(caret/circumflex)
095	01011111	_	(underscore)
096	01100000	`	
097	01100001	a	Start of Lower Case
098	01100010	b	
099	01100011	c	
100	01100100	d	
101	01100101	e	
102	01100110	f	
103	01100111	g	
104	01101000	h	
105	01101001	i	
106	01101010	j	
107	01101011	k	
108	01101100	l	
109	01101101	m	
110	01101110	n	
111	01101111	o	
112	01110000	p	
113	01110001	q	
114	01110010	r	
115	01110011	s	
116	01110100	t	
117	01110101	u	
118	01110110	v	
119	01110111	w	
120	01111000	x	
121	01111001	y	
122	01111010	z	
123	01111011	{	(left/opening brace)
124	01111100	|	(vertical bar)
125	01111101	}	(right/closing brace)
126	01111110	~	(tilde)
127	01111111	DEL	



ASCII Table

The Fact File

- **In a Non-Positional number system various different symbols are used to represent the numbers. Each symbol used in the number has the same value independent of the position of the symbol in the number. For example 1=I, 2=II, 3=III etc. The Roman number system is an example of this type**
- Disadvantage with Non-Positional number system is that arithmetic operations are difficult to perform



- **Positional number system consists of a fixed set of digits or symbols that are used to represent the numbers and the position of a digit in the number gives the value of that digit. Example decimal system**
- The base or radix of a positional number system indicates the number of different digits that are present in the number system to represent the numbers. Example: base of decimal number system is 10
- **Base of Binary number system is 2 and the digits in that system are 0, 1**
- Base of Octal number system is 8 and the digits in that system are 0, 1, 2, 3, 4, 5, 6, 7
- **Base of Hexadecimal number system is 16 and the digits in that system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F**
- Writing the base as a subscript to the number indicates the base of a number system. For example a binary number like 1001, which has a base of 2 is indicated as 1001_2
- **The Double Add Method is used to convert an integer number from binary to decimal**
- It is much easier to convert between bases which are in powers of 2, i.e. binary (2^0), Octal (2^3), and hexadecimal (2^4). To convert a value from binary to octal, note that each octal digit (0 to 7) is represented by 3 binary digits (0 → 000, 1 → 001, 2 → 010, ... 7 → 111)
- **To convert from binary to octal**

  Step1: Group the binary digits in groups of three from the right

  Step2: Replace each group by the corresponding octal digit
- To convert a value from binary to hexadecimal, each hexadecimal digit (0 to F) is represented by 4 binary digits (0 → 0000, 1 → 0001, 2 → 0010, ... E → 1110, F → 1111)

  Step1: Group the binary digits in groups of four from the right

  Step2: Replace each group by the corresponding hexadecimal digit
- **The Half Add Method is used to convert a fractional number from binary to decimal**
- The process of doing binary addition is the same as that of doing decimal addition. Binary addition is much simpler than decimal addition, as we have to deal with only two digits '0' and '1'
- **While doing addition and subtraction using octal numbers, just remember that in octal number system there are only eight digits from 0 to 7. Thus after 7 we do not have 8 but 10, 11, 12 ..., up to 17. After 17 since we do not have 18, we will have 20, 21, 22, ..., 27, 30 and so on.**
- While doing addition and subtraction using hexadecimal numbers, just remember that in hexadecimal number system there are sixteen digits from 0 to F. Thus after 9 we do not have 10 but A, B, ..., up to F. After F since we do not have any digit, we will have 10, 11, 12, ..., 1E, 1F and so on
- **Instead of using the direct method of addition/subtraction, one can use an indirect method, whereby a number in any other base is first converted to decimal and then the required calculation is done. After the calculation, the result is again converted back to the original base**
- In Sign Magnitude Representation the sign of a positive number is taken as '0' and the sign of a negative number is taken as '1'. The MSB will be used for storing the sign of the number, while the remaining bits are used for storing the magnitude of the number
- **In 1's Complement Representation a negative number is represented by the 1's complement of the positive sign magnitude value**
- To form the 1's complement of a number convert each 1 to 0 and each 0 to 1 in the positive sign magnitude number. Example, 1's complement of 0011101 is 100010
- **In 2's Complement Representation a negative number is represented by the 2's complement of the positive sign magnitude value. To get the 2's complement add 1 to the 1's complement of the positive sign magnitude value.**
- In 2's complement form there is a single representation for 0. There are no two different combinations that represent +0 and -0. Therefore 2's complement representation is a better representation method for signed numbers than the sign magnitude and 1's complement methods
- **The computer uses an additive method to do subtraction using the complement of the number to be subtracted**
- Computers use definite amount of space for storing various values
- **In fixed-point representation the decimal point is put at a fixed position within the set of available positions. This provides a fixed set of places to the left and right of the decimal point**
- If the computer uses 32 bits for representing fixed point real numbers, then we can allocate 1 bit for storing the sign of the number, 23 bits for storing the integer part, and 8 bits for storing the fraction part
- **Fixed point real numbers can be used in situations where very large numbers are not involved, or where the precision of a value is not required beyond a certain range**
- Fixed point method of representing real numbers cannot represent a wide range of values and may need a large number of digits to do so
- **The floating point representation of real numbers increases the range of numbers that can be represented using a fixed set of positions**

- **The floating point representation allows a large range of numbers to be represented using a relatively smaller number of digits by compromising digits used for precision**
- The floating point representation is based on the scientific notation for expressing both very large and very small numbers in a concise format
- **The scientific notation has three components:**
  - **Base: Indicating the base of the number system used**
  - **Exponent: Indicting the power to which the base is raised**
  - **Mantissa: Indicating number of significant digits used to represent the number**
- In a real number, the digits that are used to express the number and contribute to its precision are called the significant digits of the number
- **The number 0.00056_{10} has only 2 significant digits i.e. 5 and 6, while the remaining three '0's after the decimal point are used for determining the position of the decimal point in the number, and hence are not considered as significant digits**
- Floating point numbers are stored in a normalised form in order to avoid multiple floating point representations for the same number
- **The process of normalisation involves shifting the radix point to the left or right until the radix point is to the left of the leftmost nonzero digit. To keep the value of the number unchanged, the exponent is adjusted accordingly**
- Hence the general format for representing a normalised floating point number N is $N = sM \times R^E$
- **Precision of a floating point number is determined primarily by the number of digits in the fraction part, i.e. the mantissa M**
- Binary floating point numbers are represented as $N = sM \times 2^E$
- **One bit on the left of the mantissa is used for the sign of the mantissa. 0 for +ve, 1 for -ve**
- The number of bits to be used for the mantissa is determined by the number of significant decimal digits we wish to have for the computation. In general 23 binary digits are used to represent the mantissa
- **IEEE Standard 754 floating point number representation is the most common representation today for real numbers on computers**
- There are generally two types of numbers that are represented using IEEE standard. These are the 32 bit single precision number and the 64 bit double precision number
- **The error produced by truncating the least significant digits of a number is called round-off error**
- If two numbers in floating point representation are to be added or subtracted, the exponents of the two numbers must be made equal first. This may require shifting the radix point. To preserve the range of the numbers correctly, the radix point needs to be shifted in the number with the lesser exponent, to match the exponent of the number with the larger exponent
- **To multiply two numbers in normalised floating point representation, first multiply the corresponding mantissa of both the numbers to get the mantissa of the result. Next add the exponents of each of the numbers to get the exponent of the result**
- To divide two numbers in normalised floating point representation, first divide the mantissa of the first number with the mantissa of the second number to get the mantissa of the result. Next subtract the exponent of the second number from the exponent of the first to get the exponent of the result
- **The American Standard Code for Information Interchange (ASCII) is used for representing characters in a computer system. The representation for each character consists of 7 bits, and all 2^7 possible bit patterns represent valid characters**
- Upper case characters always contain a '0' in bit 5 and lower case alphabetic characters always contain a '1' in bit 5 position for the 7 bit ASCII code
- **The Extended Binary Coded Decimal Interchange Code (EBCDIC) is an eight-bit code that is used extensively in IBM mainframe computers and allows representing a maximum of 256 symbols**
- Indian Standard Code for Information Interchange (ISCII) is a coding scheme for representing various writing systems of India
- **Binary Coded Decimal (BCD) code is one such alphanumeric code developed by IBM and that was used to represent both digits and characters in computers. It is a 6-bit code that can be used to represent a maximum of 2^6=64 possible symbols**
- In the Excess-3 (XS-3) code developed at Harvard and used with the Mark machines, the number 0011_2 (i.e. 3_{10}) is added to each BCD value to get its equivalent Excess-3 value
- **The XS-3 code is a self-complementing code. This means that the complement of a number in the XS-3 code directly gives the corresponding 9's complement value of that number**
- XS-3 code is not a weighted code
- **In the Gray code system, only a single bit changes between any two consecutive numbers. This prevents the occurrence of any transient error values when the count changes from one value to the next**

- **Gray code is not a weighted code, as a digit in this system does not carry any positional value**
- A Gray code for a given number can be obtained by successive XOR operation between the bits
- **A digital image is basically a binary representation of a two-dimensional image**
- Depending upon the way the image is constructed, digital images can be stored either as a raster image or as a vector image
- **Raster images are composed of a set of digital values called picture elements or pixels**
- Vector images are generated by mathematical equations and require much lesser space to store
- **GIF or Graphics Interchange Format is a bitmap image format that was introduced by CompuServe in 1987 and has since come into widespread usage on the World Wide Web**
- JPEG is a commonly used method of image compression for digital images. The name JPEG stands for Joint Photographic Experts Group, which is the name of the committee that created the JPEG standard
- **TIFF or Tagged Image File Format is a file format for storing images and is popular among graphic artists, the publishing industry, and both amateur and professional photographers in general**
- PNG or Portable Network Graphics is a bitmap image format that uses lossless data compression
- **PNG offers a variety of transparency options. With truecolor and grayscale images a single pixel value can be declared as transparent, that allows a prticular background colour to become transparent**

Review Questions

Q1. **Fill in the blanks:** **1 each**

a. In a _____ number system various different symbols are used to represent the numbers.

b. _____ number system consists of a fixed set of digits or symbols that are used to represent the numbers.

c. Base of _____ number system is 2.

d. Base of _____ number system is 16.

e. The _____ Method can be used to convert an integer number from binary to decimal.

f. The _____ Method can be used to convert a fractional number from binary to decimal.

g. In hexadecimal number system there are sixteen digits from _____ to _____.

h. In Sign Magnitude Representation the sign of a positive number is taken as _____ and the sign of a negative number is taken as _____.

i. The 1's complement of 01110011 is _____.

j. The 2's complement of 01100111 is _____.

k. The 2's complement of 01101100 is _____.

l. To get the 2's complement add 1 to the _____ of the positive sign magnitude value.

m. In _____ form of signed numbers there is a single representation for 0.

n. Minimum value that can be expressed using an 'n' bit 1's complemented signed value is _____.

o. Maximum value that can be expressed using an 'n' bit sign magnitude signed number is _____.

Q2. **Multiple Choice Questions. Select from any one of the four options.** **1 each**

i) The base of a number system is:

a. The value of the maximum digit in the number system

b. The value of the minimum digit in the number system

c. The total number of distinct digits in the number system

d. The number of arithmetic operations possible on the number system

ii) The base of the number system with the digits 0, α, β, γ, μ, is:

a. 3 b. 4 c. 5 d. 6

iii) The value $(9)_{10} = (?)_{16}$

a. C b. B c. A d. 9

iv) The value $(11)_{10} = (?)_{16}$

a. 9 b. A c. B d. C

v) The value $(1111)_2 = (?)_{16}$

a. 10 b. B c. D d. F

vi) The value $(10)_8 = (?)_{16}$
a. 7 b. 8 c. 9 d. 10

vii) The value $(D)_{16} = (?)_{10}$
a. 11 b. 12 c. 13 d. 14

viii) The value $(FF)_{16} = (?)_2$
a. 11111110 b. 10000000 c. 10000001 d. 11111111

ix) The value $(11)_2 = (?)_{16}$
a. 2 b. 3 c. 4 d. 5

x) The value $(10)_2 = (?)_8$
a. 2 b. 3 c. 4 d. 5

xi) The value $(111)_2 = (?)_{16}$
a. 5 b. 6 c. 7 d. 8

xii) The value $(77)_8 = (?)_2$
a. 111110 b. 111100 c. 111111 d. 110011

xiii) The value $(10)_{16} = (?)_{10}$
a. 15 b. 16 c. 17 d. 18

xiv) The value $(10)_{10} = (?)_8$
a. 10 b. 11 c. 12 d. 13

xv) The sum of $(5)_8+(3)_8=(?)_8$
a. 8 b. 9 c. 10 d. 11

xvi) The sum of $(4)_8+(6)_8=(?)_8$
a. 10 b. 11 c. 12 d. 13

xvii) The sum of $(11)_2+(1)_2=(?)_2$
a. 101 b. 100 c. 110 d. 111

xviii) The sum of $(7)_8+(7)_8=(?)_8$
a. 14 b. 15 c. 16 d. 17

xix) The sum of $(A)_{16}+(6)_{16}=(?)_{16}$
a. F b. 10 c. 11 d. 12

xx) The sum of $(9)_{16}+(6)_{16}=(?)_{16}$
a. D b. 15 c. F d. 10

xxi) The difference $(10)_8 - (4)_8 = (?)_8$
a. 3 b. 4 c. 5 d. 6

xxii) The difference $(12)_8 - (7)_8 = (?)_8$
a. 3 b. 4 c. 5 d. 6

xxiii) The difference $(10)_{16} - (C)_{16} = (?)_{16}$
a. 3 b. 4 c. 5 d. 6

xxiv) The difference $(D)_{16} - (9)_{16} = (?)_{16}$
a. 3 b. 4 c. 5 d. 6

xxv) The difference $(111)_2 - (101)_2 = (?)_2$
a. 01 b. 00 c. 10 d. 11

xxvi) The difference $(1011)_2 - (1000)_2 = (?)_2$
a. 01 b. 00 c. 11 d. 10

xxvii) The 8 bit 1's complement representation of -110010_2 is:
a. 11001110 b. 10001101 c. 11001101 d. 00001101

xxviii) The 8 bit 1's complement representation of -100001_2 is:
a. 10011111 b. 10011110 c. 11011111 d. 11011110

xxix) The 8 bit 2's complement representation of -101110_2 is:
a. 10010001 b. 11010010 c. 11010011 d. 11010001

xxx) The 8 bit 2's complement representation of –1111012 is:
a. 00000011 b. 11111110 c. 11000011 d. 10000011

xxxi) Which method is used to convert a fractional number from binary to decimal?
a. Full Add Method b. Half Add Method c. Double Add Method d. Triple Add Method

xxxii) Which method is used to convert a whole number from binary to decimal?
a. Full Add Method b. Half Add Method c. Double Add Method d. Triple Add Method

xxxiii) Which is the best method to store signed binary numbers in a computer:
a. sign magnitude b. 2's complement c. 1's complement d. none of these

xxxiv) The full form of ASCII is:
a. American Standard Code for Input Interchange
b. American Standard Code for Information Intake
c. American Standard Code for Information Interchange
d. American Standard Code for Input Interpretation

xxxv) ASCII code is used for representing what in a computer system?
a. integers b. characters c. strings d. decimal numbers

xxxvi) The BCD representation of 23 is:
a. (00100011) b. (10001100) c. (00110010) d. (010011)

xxxvii) The BCD representation of 49 is:
a. (11100011) b. (1001001) c. (00110010) d. (01001001)

xxxviii) A single precision floating point number contains:
a. 8 bits b. 16 bits c. 64 bits d. 32 bits

xxxix) A double precision floating point number contains:
a. 16 bits b. 24 bits c. 64 bits d. 32 bits

xl) The number of significant digits in the number 0.00234 is:
a. 2 b. 3 c. 4 d. 5

xli) The number of significant digits in the number 12.2309 is:
a. 6 b. 5 c. 4 d. 3

xlii) The number of significant digits in the number 10.2300 is:
a. 3 b. 4 c. 5 d. 6

xliii) The normalised representation of the floating point value 2356.12005_{10} is:
a. 2.35612005×10^{5} b. $0.235612005 \times 10^{-4}$ c. 2.35612005×10^{3} d. $0.235612005 \times 10^{4}$

xliv) The normalised representation of the floating point value 0.00010111_2 is:
a. 0.10111×2^{3} b. 0.10111×2^{-3} c. 1.0111×2^{2} d. 1.0111×10^{3}

xlv) The normalised representation of the floating point value 0.0020050_{10} is:
a. 0.2005×10^{-2} b. 0.2005×10^{2} c. 0.20050×10^{-3} d. 2.005×10^{-2}

xlvi) The ASCII value of 'A' is:
a. 97 b. 1 c. 65 d. 122

xlvii) The ASCII value of 'a' is:
a. 97 b. 1 c. 65 d. 122

xlviii) The ASCII value of '0' is:
a. 97 b. 48 c. 65 d. 0

xlix) Which word is represented by the following ASCII values taken together [72, 73]:
a. UV b. HI c. SO d. BI

l) Which word is represented by the following ASCII values taken together [85, 86]:
a. XY b. CD c. HI d. UV

li) In a real number the digits used to express the number and contribute to its precision are called:
a. special digits b. insignificant digits c. numeric digits d. significant digits

lii) Images generated by mathematical equations are called:
a. digital images b. vector images c. raster images d. math images

liii) Images composed of a set of digital values called picture elements or pixels are called:
a. raster images b. vector images c. pixel images d. digital images

liv) ISCII stands for:
a. Indian Special Code for Information Interchange
b. Indian Standard Code for Input Information
c. Indian Standard Code for Information Interpretation
d. Indian Standard Code for Information Interchange

lv) Which non-weighted binary code uses XOR operation to form the code?
a. ASCII code b. BCD code c. 2421 Code d. Gray code

lvi) The Gray code for the binary number $(101)_2$ is?
a. 101_2 b. 111_2 c. 010_2 d. 110_2

lvii) EBCDIC stands for:
a. The Extended Binary Coded Decimal Interchange Code
b. The Extended Binary Coded Decimal Interchange Code
c. The Extended Binary Coded Decimal Interchange Code
d. The Extended Binary Coded Decimal Interchange Code

Q3. **Short Answer type questions:** **1 each**

i) What do you mean by the radix of a number system?

ii) What is the radix of a number system which uses the digits 0, α, β, λ, and π?

iii) What is the value of the operation (π - β) of a number system which uses the digits 0, α, β, λ, π and δ?

iv) What is the value of the operation (α + β) of a number system which uses the digits 0, α, β, λ, π, and δ?

v) What is the maximum difference possible between two digits of a number system which uses the digits 0, α, β, λ, π, and δ?

vi) What is the maximum value possible using three digits for a number system which uses the digits 0, α, β, λ, π, and δ?

vii) How many digits are there in the hexadecimal number system?

viii) State one advantage of using a positional number system over a non-positional one.

ix) How many binary digits are used to represent an octal digit?

x) How many binary digits are used to represent a hexadecimal digit?

xi) How many binary digits will be required to represent a digit in a number system which has a total of 2^k digits in it?

xii) What are the digits used in the hexadecimal number system?

xiii) Explain the meaning of the value 444_{10} with respect to the position of the digits.

xiv) Express the binary value -10110_2 in 8 bit 1's complement form.

xv) Express the binary value -111_2 in 8 bit 2's complement form.

xvi) State one advantage of using 2's complement representation for signed binary numbers over 1's complement representation.

xvii) What is the maximum and minimum value that can be obtained when adding two octal digits?

xviii) Write the full form of BCD.

xix) Write the full form of EBCDIC.

xx) Write the full form of ASCII.

xxi) Write the full form of ISCII.

xxii) State one difference between vector and raster graphics.

xxiii) Write the extensions used for any two graphic data formats.

xxiv) What is the advantage of using Gray code over Excess-3 code?

xxv) Name a self complementing binary code.

xxvi) What do you mean by the significant digits of a floating point number?

xxvii) How many significant digits are there in the floating point number 0.0010060040_{10}?

xxviii) What is the normalised form of the decimal number 0.000120060_{10}?

xxix) State one difference between fixed and floating point number representation.

xxx) Write the ASCII values for the alphabets 'A' and 'a'.

LA

Q4. **Long Answer type questions:** **7 each**

i) Convert the following from Decimal to Binary representation: 2+2
a. $(255)_{10}$ b. $(1024)_{10}$
Carry out the following subtraction using 2's complement method: $(110110)_2 - (10110)_2$ 3

ii) Convert the following from Decimal to Binary representation: 2+2
a. $(513)_{10}$ b. $(20459)_{10}$
Carry out the following subtraction using 1's complement method: $(101001)_2 - (111100)_2$ 3

iii) Convert the following from Decimal to Octal representation: 2+2
a. $(402)_{10}$ b. $(4096)_{10}$
Carry out the following multiplication: $(100111)_2 \times (1011)_2$ 3

iv) Convert the following from Decimal to Octal representation: 2+2
a. $(1001)_{10}$ b. $(2049)_{10}$
Carry out the following multiplication: $(101100)_2 \times (1101)_2$ 3

v) Convert the following from Decimal to Hexadecimal representation: 2+2
a. $(2048)_{10}$ b. $(5600)_{10}$
Carry out the following division: $(1111011100)_2 \div (10011)_2$ 3

vi) Convert the following from Decimal to Hexadecimal representation: 2+2
a. $(1024)_{10}$ b. $(6521)_{10}$
Carry out the following division: $(1001011)_2 \div (1111)_2$ 3

vii) a. Convert $(1234)_{10}$ to binary. 2
b. Convert $(0.125)_{10}$ to octal. 2
c. Find the value of $110110_2 \times 1011_2$ 3

viii) a. Convert $(1AB)_{16}$ to decimal. 2
b. Convert $(0.1101101)_2$ to octal. 2
c. Find the value of $(1001011)_2 \div (1111)_2$ 3

ix) a. Convert $(C0AB)_{16}$ to octal. 2
b. Convert $(0.26)_{10}$ to hexadecimal. 2
c. Find the value of $(1001001001)_2 \div (1101)_2$ 3

x) a. Convert $(2540)_8$ to hexadecimal. 2
b. Convert $(0.11101)_2$ to decimal. 2
c. Find the value of $(100100)_2 \times (1101)_2$ 3

xi) a. Convert $(1001)_{16}$ to decimal. 2
b. Convert $(0.AD)_{16}$ to octal. 2
c. Find the value of $(10010)_2 \times (111)_2$ 3

xii) a. Convert $(265.55)_{10}$ to octal. 2+2
b. Find the value of $(10110)_2 - (1101)_2$ 2
c. Find the 1's complement of $(101100)_2$ in 8 bit form. 1

xiii)	a. Convert $(1A7.BD3)_{16}$ to octal.	2+2
	b. Find the value of $(573)_8$ - $(246)_8$	2
	c. Find the 1's complement of $(110011)_2$ in 8 bit form.	1
xiv)	a. Convert $(456.123)_8$ to hexadecimal.	2+2
	b. Find the value of $(A57)_{16}$ - $(42D)_2$	2
	c. Find the 1's complement of $(111110)_2$ in 8 bit form.	1
xv)	a. Convert $(101101101.11110001)_2$ to octal.	2+2
	b. Find the value of $(465)_8$ - $(366)_8$	2
	c. Find the 1's complement of $(10011)_2$ in 8 bit form.	1
xvi)	a. What is the value of $(234)_5$ in base 10?	2
	b. Subtract the following using 1's complement method: $(1100111)_2 - (1011011)_2$	3
	c. Convert $(ABCD)_{16}$ to octal.	2
xvii)	a. What is the value of $(1221)_3$ in base 10?	2
	b. Subtract the following using 2's complement method: $(1101011)_2 - (1011111)_2$	3
	c. Convert $(23456)_8$ to hexadecimal.	2
xviii)	a. What is the value of $(2112)_3$ in base 4?	4
	b. Subtract the following using 2's complement method: $(10110100)_2 - (11001)_2$	3
xix)	a. What is the value of $(333)_4$ in base 3?	4
	b. Subtract the following using 2's complement method: $(10110100)_2 - (11001)_2$	3
xx)	a. What is the value of $(2222)_4$ in base 5?	4
	b. Subtract the following using 1's complement method: $(1011011)_2 - (1100110)_2$	3
xxi)	a. Add the following: $(234)_8 + (537)_8$	2
	b. Subtract the following using 2's complement method: $(10110100)_2 - (11001)_2$	3
	c. Find the value of $(ABC)_{16}$ in base 8.	2
xxii)	a. Add the following: $(110011001)_2 + (101100110)_2$	2
	b. Subtract the following using 1's complement method: $(1100101)_2 - (1111011)_2$	3
	c. Find the value of $(0.26)_{10}$ in base 8.	2
xxiii)	a. Add the following: $(A8D)_{16} + (5B8)_{16}$	2
	b. Multiply the following: $(1100101)_2 \times (1101)_2$	3
	c. Find the value of $(0.11011)_2$ in base 10.	2
xxiv)	a. Add the following: $(5236)_8 + (555)_8$	2
	b. Subtract the following: $(10110)_2 - (10011)_2$	2
	c. Divide the following: $1100010000_2 \div 1110_2$	3
xxv)	a. Add the following: $(AD9)_{16} + (DD7)_{16}$	2
	b. Subtract the following: $(653)_8 - (537)_8$	2
	c. Multiply the following: $11001101_2 \times 1011_2$	3
xxvi)	a. Add the following: $(D02B)_{16} + (AC7)_{16}$	2
	b. Multiply the following: $(110110)_2 \times (1011)_2$	2
	c. Divide the following: $1101101010_2 \div 10011_2$	3
xxvii)	a. Find the following using 1's complement subtraction method: $(1101101)_2 - (1111000)_2$	3
	b. Find the value of $(ABCD.EF12)_{16}$ in base 8	2+2
xxviii)	a. Find the following using 2's complement subtraction method: $(101101)_2 - (1100110)_2$	3
	b. Find the value of $(12345.6712)_8$ in base 16	2+2
xxix)	a. Find the value of x in the following expression: $(56)_8 + (1100110)_2 = (x)_{16} + (42)_{10}$	5
	b. Multiply the following: $(1010)_2 \times (101)_2$	2

xxx) a. Find the value of y in the following expression: $(98)_{10} + (y)_2 = (AB)_{16} + (20)_8$ 5
b. Divide the following: $101101_2 \div 101_2$ 2

xxxi) a. Perform the floating point operation 0.1234 E-3 + 0.3456 E-2 2
b. Perform the floating point operation 0.7896 E+20 - 0.5876 E+22 2
c. What do you mean by significant digits of a floating point number? 1
d. What do you mean by the mantissa and exponent of a floating point number? 2

xxxii) a. Perform the floating point operation $0.1101 \times 2^{34} + 0.1011 \times 2^{32}$ 2
b. Perform the floating point operation $0.1101 \times 2^{-3} - 0.1011 \times 2^{-4}$ 2
c. Explain round-off error. 2
d. What do you mean by a fixed point number? 1

xxxiii) a. Perform the floating point operation (0.7896 E+22) ÷ (0.5876 E+20) 2
b. Perform the floating point operation $(0.1021 \times 10^{05}) \times (0.9677 \times 10^{08})$ 2
c. What do you mean by the range and precision of a floating point number? 2
d. How many significant digits are there in the number 09.000980? 1

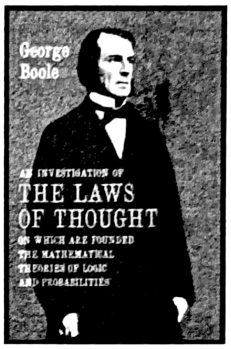
xxxiv) a. Perform the floating point operation $(0.1101 \times 2^{6}) \times (0.1001 \times 2^{8})$ 2
b. Perform the floating point operation $(0.1101 \times 2^{-4}) \div (0.1011 \times 2^{-4})$ 2
c. Explain the advantage of using a floating point number over a fixed point number. 2
d. What is the normalised form of the floating point number $-2464.009701 \times 10^{3}$? 1

CHAPTER 4
Boolean Algebra and Logic Gates

- **Basic Boolean Operations** 4-1
- **OR, AND, NOT Operations and Truth Tables** 4-2
- **Switching Circuit Equivalents** 4-3
- **Boolean Algebra Rules and Proof by Perfect Induction** 4-3
- **De' Morgan's Theorems & Basic Duality of Boolean Rules** 4-8
- **SOP, POS, Min Term & Max Term Expressions** 4-11
- **Canonical forms of Boolean Expressions and their Complements** 4-15
- **Techniques for Simplification: Using Karnaugh Map** 4-19
- **Logic Gates: AND, OR, NOT, XOR, NAND, NOR, XNOR** 4-26
- **NAND and NOR Gates as Universal Logic Gates** 4-27
- **Worked out problems on Logic Gates** 4-31

4.1 Basic Boolean Operations

Boolean algebra was developed by the English mathematician **George Boole**. His work was based on the analysis of logic and is contained in his book named "*An investigation of the laws of thought on which are founded the Mathematical Theories of Logic and Probabilities*", published in 1854. His ideas deal with a **mathematical analysis of logic**. In Boolean algebra first a set of initial assumptions are made called postulates. Based on these postulates next a truth table is prepared. The table shows the relation between all the possible input logic possibilities and the corresponding output logic.



The logic we apply in studying digital systems is based on a definition of Truth made popular by Greek philosopher Aristotle (384 BC – 322 BC)

A **valid** logical expression is denoted as **TRUE** and is assigned a numeric value of '**1**'. Similarly an **invalid** logical expression is denoted as **FALSE** and is assigned a numeric value of '**0**'. For example the statement "*Sun rises in the West*" is logically false and hence is assigned a logical value of '0'. Similarly the statement "*Water flows from a high level to a low level*" is logically true and hence is assigned a logical value of '1'.

Thus **in Boolean algebra there are only two kinds of states FALSE and TRUE that a variable can take** and are denoted by '0' and '1'. This is unlike normal algebra where a variable may assume a range of values. For example in the relation $y = x^2$, y can assume any value from 0 to $+\infty$ for real values of x. Accordingly the rules of Boolean algebra are a little different form those of traditional algebra.

But **why do we need to study Boolean algebra**? All modern day computers are electronic devices that are able to do arithmetic and logical operations. Proper electronic circuits are there in a computer to do these operations. These logical operations are done using the two basic logic states **True and False.** These states are **represented in the circuit by electronic switches**. The **ON** state of such a switch represents a true logic and the **OFF** state represents a false logic.

The different postulates of Boolean algebra are based on the two logic states True and False

The different postulates of Boolean algebra based on the two logic states **True** and **False** serve as a design tool for these logic circuits. It helps to form the blueprint for the design of the logic circuits.

Apart from the variables used to represent the logic states, there are certain **basic operations that can be performed** on these variables. These basic operations include **OR, AND**, and the **COMPLEMENT** operation. There are other operations also, but these are various combinations of these three basic operations.

The following examples will make these basic operations clear and will give an understanding of how Boolean algebra works. Let us put down in a **tabular manner** the different possibilities that can arise for a person to sit in a bus seat where it is written "**Seat for Senior Citizen OR Disable Persons**".

Senior Citizen	Disable Person	Should Sit
No *(False,0)*	**No** *(False,0)*	**No** *(False,0)*
No *(False,0)*	**Yes** *(True,1)*	**Yes** *(True,1)*
Yes *(True,1)*	**No** *(False,0)*	**Yes** *(True,1)*
Yes *(True,1)*	**Yes** *(True,1)*	**Yes** *(True,1)*

From the table shown on the right we find that **there are 4 different possibilities**. If we read the table row-wise, we get the four possibilities. For example, we find that if a person is neither senior nor disabled, then he should not be sitting on that seat. This is represented in the first row of the table. Similarly if a person is both senior and disabled then also he can use the seat. This is represented in the last row of the table. If the person is either senior or disabled then also he should be offered the seat. These are indicated in the second and third rows of the table.

About 2000 years after Aristotle, George Boole (1815 – 1864) turned Aristotle's logic into an Algebra. 90 years later, a Master's student at MIT, Clause Shannon, proved that Boole's algebra could be applied to electrical systems.

In the table we have **indicated these 4 possibilities** by a **Yes** or a **No**. We find that **if either of the conditions is satisfied, then the person is allowed to sit** in that particular seat. Note that the logic False or '0' can also be used in place of a No, and the logic True or '1' can be used in place of a Yes.

Next let us put down in a tabular manner the different possibilities for a person **to write using pen and paper**.

Pen	Paper	Write
No *(False,0)*	**No** *(False,0)*	**No** *(False,0)*
No *(False,0)*	**Yes** *(True,1)*	**No** *(False,0)*
Yes *(True,1)*	**No** *(False,0)*	**No** *(False,0)*
Yes *(True,1)*	**Yes** *(True,1)*	**Yes** *(True,1)*

From the table shown on the right we find there are 4 different options. However, if **both pen and paper are present** then only you can write. It is not possible to write with only pen or with only paper or neither. Accordingly, only the last row of the table has a **Yes** in the output. All other outputs are **No**.

Next consider a **street light**. When it is morning, a street light is off and at night it is on. If this is put in the form of a table then what we get is shown on the right.

Morning	Street Light On
No *(False,0)*	**Yes** *(True,1)*
Yes *(True,1)*	**No** *(False,0)*

A logical OR operation is True if at least one of the conditions involved is True

A logical AND operation is True if and only if all the conditions involved are True

A logical NOT operation gives the inverse of a given logic

The **first example** represents a logical situation known as **OR Logic** and the **second example** represents a logical situation known as **AND Logic**. Usually the tables are written using only '0's and '1's, representing the logic states False and True respectively. The resultant tables are shown on the right.

OR Logic

Old	Disable	Sit
0	0	0
0	1	1
1	0	1
1	1	1

AND Logic

Pen	Paper	Write
0	0	0
0	1	0
1	0	0
1	1	1

The **third basic operation is the complement operation**. It is basically the **inverse** of a particular logic. Thus the complement of the statement "The person is tall" is "The person is **NOT** tall". By replacing the words with the symbols '0' and '1' what we find from the table is that the **output logic is the reverse of the input**.

COMPLEMENT / NOT Logic

Morning	Street Light On
0	1
1	0

4.2 OR, AND, NOT Operations and Truth Tables

The results of the above examples can be generalised and **written in the form of a mathematical relation**. This helps in easy manipulation of the results and writing complex results that involve more than one logical operation. The results are written using symbols in a tabular manner known as a **Truth Table**. **Variable names are used** to represent a particular situation. For example the condition "Old Man" can be represented by the variable symbol **A** and the condition "Disabled Man" can be represented by the variable symbol **B**. The **logical operations are also expressed using special symbols**.



OR Logic

- OR Logic

OR Logic

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

The **OR** operation is denoted by the **'+' operator** and is used to indicate a **Logical Addition**. The output of a logical OR operation is True if at least one of the conditions involved in the operation is True. The logical output is written as:

C = A+B and is read as **"C equals A OR B"** or **"C equals A plus B"**.

The truth table for a **2 variable OR operation** is shown on the right. Note that for a logical OR operation the **output is '0' only when both the inputs are '0'**, otherwise the output is '1'. Also note that 1+1=1 and not 2 for a logical addition.



AND Logic

- AND Logic

AND Logic

A	B	A·B
0	0	0
0	1	0
1	0	0
1	1	1

The **AND** operation is designated by the '·' (dot) **operator** and is used to indicate a **Logical Multiplication**. The output of a logical AND operation is True if and only if all the conditions involved in the operation are True. The logical output is written as:

C = A·B and is read as **"C equals A AND B"** or **"C equals A B"**.

The truth table for a **2 variable AND operation** is shown in the previous page. **Note** that the '.' is often dropped and the expression written as **C = AB**. Also, for a logical AND operation the **output is '0' if any one of the inputs is a '0'.**

Note that **logical AND has a higher priority over logical OR operation** just as with traditional multiplication and addition. Thus in an expression if there is both an AND and an OR operation, then the AND operation has to be done first. However, this priority can be changed by using brackets.

Boolean algebra is used in information theory as almost all search engines allow someone to enter queries in the form of logical expressions.

- NOT Logic

The **NOT** or complement operation is designated by a ' ¯ ' (bar) placed over the variable and is used to indicate a **Logical Complement**. It is a unary operation i.e. needs a minimum of one variable to act upon. The logical output is written as:

C = $\bar{A}$ and is read as **"C is the complement of A"**

NOT Logic

A	$\bar{A}$
0	1
1	0

The truth table for a **single variable NOT operation** is shown on the right. **Note** that while writing a NOT operation sometimes the apostrophe mark is also used as **C = A'** which is same as **C = $\bar{A}$.**

4.3 Switching Circuit Equivalents

Another area where the ideas of Boolean algebra are used is in the design of switching circuits as illustrated below. In *figure 1* we have **two switches A & B in parallel** connected to a calling bell. **V** is a voltage source which drives the circuit. It can be seen that if any one of the switches is **ON** the bell will ring. Even if both the switches are **ON** then also the bell will ring. **Only when both the switches are OFF, the bell will not ring.** Thus the above circuit functions like **OR** logic with the output ON (bell ringing) whenever any one or both the switches are ON.

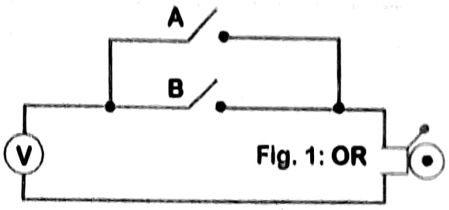


Fig. 1: OR

Switching Circuit OR operation

In *figure 2* the **switches A & B are connected in series.** If any one or both the switches are **OFF** the bulb will not glow. Only when **both the switches are ON will the bulb glow.** This is similar to **AND** logic where the output is high only when both inputs are high. A bed-switch operates in this principle. Both the main switch and the bedside switch should be ON to put on the light.

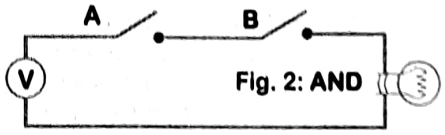


Fig. 2: AND

Switching Circuit AND operation

Figure 3 shows the electrical equivalent of NOT logic. This is a burglar alarm circuit (though not a good one) where the alarm rings whenever the door D is opened. When the door is closed, the current maintained by the voltage source **V** flows through the door circuit with minimum resistance and no current flows through the alarm circuit and hence the alarm is in an off state. When the door is opened the only path for the current is through the alarm and the alarm starts ringing. For the door closed (1 state) the alarm is **OFF** (0 state) and for the door open (0 state) the alarm is **ON** (1 state). Hence the **output state is the inverse of the input state** as in **NOT** operation.

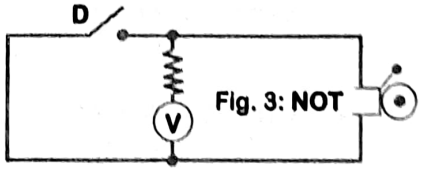


Fig. 3: NOT

Switching Circuit NOT operation

Hence we have seen that functionally the **switching circuits work like logic elements** and along with relays, these constituted the main working logic of the computers of the twenties and thirties.

4.4 Boolean Algebra Rules and Proof by Perfect Induction

As we have stated earlier, logical expressions when put in a mathematical form constitute Boolean expressions. To simplify complex Boolean expressions several rules have been developed. The rules for a two element (0, 1) Boolean algebra, also known as **switching algebra** are given in the next page. We will first prove some of the rules and then try to simplify Boolean expressions using those rules.

Boolean Rules

To start with, remember that a **NOT operation has the highest priority** followed by AND and OR operations. In an expression containing multiple such operators, **execute all Complement/NOT operations first, followed by AND operations, followed by OR operations.**

A logical AND has a higher priority than a logical OR operation

Boolean Rules

- **Boolean algebra / Switching algebra Rules**

Variables	Rules	
1	$X+0 = 0+X = X$ $X+1 = 1+X = 1$ $X+X = X$ (*Idempotent Law*) $X+\bar{X} = \bar{X}+X = 1$ $\bar{\bar{X}}=X$	$X\cdot 0 = 0\cdot X = 0$ $X\cdot 1 = 1\cdot X = X$ $X\cdot X = X$ (*Idempotent Law*) $X\cdot\bar{X} = \bar{X}\cdot X = 0$
2	$X+Y = Y+X$ (*Commutative Law*) $X+XY = X$ (*Absorptive Law*) $X+\bar{X}Y = X+Y$	$X\cdot Y = Y\cdot X$ (*Commutative Law*) $X\cdot(X+Y) = X$ (*Absorptive Law*)
3	$X+(Y+Z) = (X+Y)+Z$ (*Associative Law*) $X\cdot Y+X\cdot Z = X\cdot(Y+Z)$ (*Distributive Law*)	$X\cdot(Y\cdot Z) = (X\cdot Y)\cdot Z$ (*Associative Law*) $(X+Y)\cdot(X+Z) = X+Y\cdot Z$ (*Distributive Law*)

The above rules can be proved **using truth tables**. Since Boolean variables can assume only two values i.e. 0 & 1, we prove the rules to be true for these two values only. The proofs for rules of a single variable are given below in a tabular form.

Proof of basic Boolean Rules

X	$\bar{X}$	X+0=X	X·1=X	X+1=1	X·0=0	X+X=X	X·X=X	$X+\bar{X}=1$	$X\cdot\bar{X}=0$
0	**1**	0+0=0	0·1=0	0+1=1	0·0=0	0+0=0	0·0=0	0+1=1	0·1=0
1	**0**	1+0=1	1·1=1	1+1=1	1·0=0	1+1=1	1·1=1	1+0=1	1·0=0

Idempotent Rule: For a Boolean variable A, $A+A=A$, $A\cdot A=A$

For a Boolean variable A, $A+1=1$, $A+\bar{A}=1$

The **Commutative, Associative** and **Distributive laws** are the **postulates or axioms** based on which Boolean algebra has been developed along with the single variable rules (other rules or laws will be proved using the above laws) and hence need not be proved. However the above rules can be proved by other methods using Venn diagrams or by perfect induction. The student may try these methods if interested. The results of the remaining rules are unlike traditional algebra and we will prove them using the previous rules (these can also be proved using truth tables as above as discussed later).

- **$X+XY$** $= X(1+Y)$
  $= \mathbf{X}$ *{since $1+Y = 1$}*

- **$X(X+Y)$** $= XX + XY$
  $= X + XY$ *{since $X\cdot X = X$}*
  $= X(1+Y)$
  $= \mathbf{X}$ *{since $1+Y = 1$}*

- **$(X+Y)(X+Z)$** $= X(X+Z) + Y(X+Z)$
  $= XX + XZ + YX + YZ$
  $= X + XZ + XY + YZ$ *{since $XX=X$ and $YX=XY$}*
  $= X(1+Z+Y) + YZ$
  $= \mathbf{X + YZ}$ *{since $1+A=1$ and here $A = Z+Y$}*

- **$X+\bar{X}Y$** $= X(1+Y) + \bar{X}Y$ *{since $1+Y = 1$}*
  $= X + XY + \bar{X}Y$
  $= X + (X+\bar{X})Y$
  $= X + 1.Y$ *{since $X+\bar{X} = 1$}*
  $= \mathbf{X + Y}$

Examples of simplification

Example-1: Simplify the Boolean expression $Y\,Z\,(X\,\bar{Y}\,Z + \bar{X}\,\bar{Y}\,Z + \bar{X}\,Y\,Z)$

The expression $= Y\,Z\,Z\,(X\,\bar{Y} + \bar{X}\,\bar{Y} + \bar{X}\,Y)$ *{taking Z common}*

$= Y\,Z\,((X + \bar{X})\,\bar{Y} + \bar{X}\,Y)$ *{as ZZ = Z & taking $\bar{Y}$ common}*

$= Y\,Z\,(1.\,\bar{Y} + \bar{X}\,Y)$ *{as $X + \bar{X} = 1$}*

$= Y\,Z\,\bar{Y} + Y\,Z\,\bar{X}\,Y$

$= 0 + Z\,\bar{X}\,Y$ *{as $Y\,\bar{Y} = 0$ and $Y\,Y = Y$}*

$= \bar{X}\,Y\,Z$

Example-2: Simplify the Boolean expression $A\,B + A\,\bar{B} + \bar{A}\,C + \bar{A}\,\bar{C}$

The expression $= A\,B + A\,\bar{B} + \bar{A}\,C + \bar{A}\,\bar{C}$

$= A\,(B + \bar{B}) + \bar{A}\,(C + \bar{C})$ *{taking A and $\bar{A}$ as common}*

$= A + \bar{A}$ *{as $X + \bar{X} = 1$}*

$= 1$ *{as $X + \bar{X} = 1$}*

Example-3: Simplify the Boolean expression $A\,\bar{B}\,\bar{C} + \bar{A}\,\bar{B}\,\bar{C} + \bar{A}\,B\,\bar{C} + \bar{A}\,\bar{B}\,C$

The expression $= \bar{A}\,\bar{B}\,\bar{C} + A\,\bar{B}\,\bar{C} + \bar{A}\,\bar{B}\,\bar{C} + \bar{A}\,B\,\bar{C} + \bar{A}\,\bar{B}\,\bar{C} + \bar{A}\,\bar{B}\,C$ *{by adding $\bar{A}\,\bar{B}\,\bar{C}$ to every term}*

$= (\bar{A} + A)\,\bar{B}\,\bar{C} + (\bar{B} + B)\,\bar{A}\,\bar{C} + (\bar{C} + C)\,\bar{A}\,\bar{B}$

$= \bar{B}\,\bar{C} + \bar{A}\,\bar{C} + \bar{A}\,\bar{B}$ *{as $X + \bar{X} = 1$}*

In the above example it can be seen that we have repeatedly added the term $\bar{A}\,\bar{B}\,\bar{C}$ in the first step. However, the value of the expression remains unchanged because of the rule X + X = X. In this case we have taken X as $\bar{A}\,\bar{B}\,\bar{C}$, which is a term in the given expression.

Example-4: Simplify $C(A+\bar{B})+B+A\bar{C}$

The expression $= CA+C\bar{B}+B+A\bar{C}$

$= AC+A\bar{C}+C\bar{B}+B$

$= A(C+\bar{C})+C\bar{B}+B$

$= A(1)+C\bar{B}+B$ *{as $X + \bar{X} = 1$}*

$= A+B+\bar{B}C$

$= A+B+C$ *{as $X + \bar{X}Y = X+Y$}*

Example-5: Simplify the following Boolean expression: $(\bar{X}+XY+\bar{Y})(\bar{Y}+YZ+\bar{Z})(\bar{Z}+ZX+\bar{X})$

The expression $= (\bar{X}+XY+\bar{Y})(\bar{Y}+YZ+\bar{Z})(\bar{Z}+ZX+\bar{X})$

$= (\bar{X}+Y+\bar{Y})(\bar{Y}+Z+\bar{Z})(\bar{Z}+X+\bar{X})$ {since $A+\bar{A}B = A+B$ and here $A=\bar{X}$, i.e. $(\bar{X}+XY) = \bar{X}+\bar{\bar{X}}Y = [\bar{X}]+[\overline{\bar{X}}]\,Y = \bar{X}+Y$ }

$= (\bar{X}+1)(\bar{Y}+1)(\bar{Z}+1)$ *{as $A + \bar{A} = 1$}*

$= 1\cdot1\cdot1$ *{since A+1=1}*

$= 1$

Example-6: Simplify the following Boolean expression: $(\bar{A}+B)(\bar{B}+C)(\bar{C}+A)(\bar{B}C+\bar{A})(\bar{C}A+\bar{B})(\bar{A}B+\bar{C})$

The expression $= (\bar{A}+B)(\bar{B}+C)(\bar{C}+A)(\bar{B}C+\bar{A})(\bar{C}A+\bar{B})(\bar{A}B+\bar{C})$

$= (\bar{A}+B)(\bar{B}C+\bar{A})(\bar{B}+C)(\bar{C}A+\bar{B})(\bar{C}+A)(\bar{A}B+\bar{C})$ *{rearranging the terms}*

$= (\bar{A}+B)(\bar{A}+\bar{B}C)(\bar{B}+C)(\bar{B}+\bar{C}A)(\bar{C}+A)(\bar{C}+\bar{A}B)$ *{rearranging the terms}*

$= (\bar{A}+B\bar{B}C)(\bar{B}+C\bar{C}A)(\bar{C}+A\bar{A}B)$ *{since $(X+Y)(X+Z)=X+YZ$}*

$= (\bar{A}+0)(\bar{B}+0)(\bar{C}+0)$ *{since $X\,\bar{X} = 0$}*

$= \bar{A}\,\bar{B}\,\bar{C}$

Example-7: Show that the Boolean expression $ABC+AB\bar{C}+\bar{A}BC+A\bar{B}C+A+B+C$ is equal to $A+B+C$

The expression = $ABC + AB\bar{C} + \bar{A}BC + A\bar{B}C + A + B + C$

= $ABC + AB\bar{C} + ABC + \bar{A}BC + ABC + A\bar{B}C + A + B + C$ *{as $X + X = X$, here $X = ABC$}*

= $AB(C + \bar{C}) + BC(A + \bar{A}) + AC(B + \bar{B}) + A + B + C$

= $AB(1) + BC(1) + AC(1) + A + B + C$ *{as $X + \bar{X} = 1$}*

= $AB + BC + AC + A + B + C$

= $A + AB + B + BC + C + AC$ *{rearranging the terms}*

= $A(1+ B) + B(1 + C) + C(1 + A)$

= $A.1 + B.1 + C.1$ *{since $1+X = 1$}*

= $A + B + C$ (Proved)

Example-8: Simplify the following Boolean expression: $(X+Y)(Y+Z)(Z+X) + X + YZ$

The expression = $(X+Y)(Y+Z)(Z+X) + X + YZ$

= $(X+Y)(X+Z)(Y+Z) + X + YZ$ *{rearranging the term $(Z+X)$}*

= $(X+YZ)(Y+Z) + (X + YZ)$ *{as $(A+B)(A+C)=A+BC$}*

= $(X+YZ)[(Y+Z) + 1]$

= $(X+YZ) \cdot 1$ *{as $A+1=1$}*

= $X+YZ$

Example-9: Simplify the following Boolean expression: $(A+B+\bar{C})(A+B+C)(\bar{A}+B)\bar{B}$

The expression = $(A+B+\bar{C})(A+B+C)(\bar{A}+B)\bar{B}$

= $[\mathbf{(A+B)}+\bar{C})][\mathbf{(A+B)}+C)](\bar{A}+B)\bar{B}$ *{taking $(A+B)$ as X, $\bar{C}$ as Y, C as Z, and using $(X+Y)(X+Z)=X+YZ$ }*

= $[(A+B)+\bar{C}C](\bar{A}+B)\bar{B}$

= $(A+B)(\bar{A}+B)\bar{B}$ *{as $X\bar{X} = 0$}*

= $(B+A)(B+\bar{A})\bar{B}$

= $(B+A\bar{A})\bar{B}$ *{as $(X+Y)(X+Z) = X+YZ$}*

= $(B+0)\bar{B}$ *{as $X\bar{X} = 0$}*

= $B\bar{B}$

= 0 *{as $X\bar{X} = 0$}*

Example-10: Simplify the following Boolean expression: $\bar{A} + (A+\bar{B}+C)(A+B+\bar{C})$

The expression = $\bar{A} + (A+\bar{B}+C)(A+B+\bar{C})$

= $\bar{A} + [A + (\bar{B}+C)][A + (B+\bar{C})]$

= $\bar{A} + [A + (\bar{B}+C)(B+\bar{C})]$ *{applying the rule $(X+Y)(X+Z)=X+YZ$}*

= $\bar{A} + A + (\bar{B}+C)(B+\bar{C})$

= $(\bar{A} + A) + (\bar{B}+C)(B+\bar{C})$

= $1 + (\bar{B}+C)(B+\bar{C})$ *{as $X + \bar{X} = 1$}*

= 1 *{as $1+X=1$, and $X = (\bar{B}+C)(B+\bar{C})$}*

Example-11: Simplify the Boolean expression $A(A+B+C)(\bar{A}+B+C)(A+\bar{B}+C)(A+B+\bar{C})$

The expression = $A(A+B+C)(A+B+C)(A+B+C)(\bar{A}+B+C)(A+\bar{B}+C)(A+B+\bar{C})$ *{since $X \cdot X = X$ and here $X=(A+B+C)$}*

= $A[(A+B+C)(\bar{A}+B+C)][(A+B+C)(A+\bar{B}+C)][(A+B+C)(A+B+\bar{C})]$

= $A[(B+C+A)(B+C+\bar{A})][(A+C+B)(A+C+\bar{B})][(A+B+C)(A+B+\bar{C})]$ *{After rearranging}*

= $A[\{\mathbf{(B+C)}+A\}\{\mathbf{(B+C)}+\bar{A}\}][\{\mathbf{(A+C)}+B\}\{\mathbf{(A+C)}+\bar{B}\}][\{\mathbf{(A+B)}+C\}\{\mathbf{(A+B)}+\bar{C}\}]$

$= A[(B+C)+A\bar{A}][(A+C)+B\bar{B}][(A+B)+C\bar{C}]$ *{since (X+Y)(X+Z)=X+YZ}*

$= A[(B+C)+0][(A+C)+0][(A+B)+0]$ *{since $X\bar{X}=0$}*

$= A(B+C)(A+C)(A+B)$

$= A(A+C)(A+B)(B+C)$ *{rearranging}*

$= A(A+BC)(B+C)$ *{since (X+Y)(X+Z)=X+YZ}*

$= (AA+ABC)(B+C)$

$= (A+ABC)(B+C)$ *{since XX = X}*

$= A(1+BC)(B+C)$

$= A(B+C)$ *{since 1+X = 1}*

- **Proof using Perfect Induction method**

Proof using Perfect Induction

To prove the equivalence of the two sides of a Boolean expression, apart from the algebraic proof, there is another method called **proof by perfect induction.** It uses the truth table of a Boolean expression.

The different values that a particular Boolean variable can assume are put in a tabular manner. Then each term in the expression is evaluated for each value of the variable on both sides of the equal sign. If the final results or values on either side of the equal sign are identical, then the expressions are said to be equal. **In this method every term in the relation needs to be evaluated step by step** to get the final outputs on the either side. The following examples will make the method clear.

Proof by Perfect Induction uses the truth table of a Boolean expression.

Example-12: Prove by the method of perfect induction: $X+\bar{X}Y = X+Y$

1	2	3	4	5	6
X	Y	$\bar{X}$	$\bar{X}Y$	$X+\bar{X}Y$	X+Y
0	0	1	0	0	0
0	1	1	1	1	1
1	0	0	0	1	1
1	1	0	0	1	1

In the table shown on the left, columns 1 and 2 contain the different combination of values that the variables X and Y can take. For two variables, there will be $2^2=4$ combinations as shown in the table.

Columns 3 and 4 generate the different terms in the left side of the expression. Column 5 generates the expression in the left hand side of the equal sign by combining the terms from columns 3 and 4. Column 6 finally generates the term on the right hand of the equal sign.

From the columns 5 and 6 we find that both the columns are identical in value. Hence the expression is proved to be true for all the combinations of X and Y, by perfect induction.

Example-13: Prove by the method of perfect induction: $XY(\bar{X}Y\bar{Z}+X\bar{Y}\bar{Z}+\bar{Z}) = XY\bar{Z}$

1	2	3	4	5	6	7	8	9	10	11	12
X	Y	Z	$\bar{X}$	$\bar{Y}$	$\bar{Z}$	XY	$\bar{X}Y\bar{Z}$	$X\bar{Y}\bar{Z}$	$\bar{X}Y\bar{Z}+X\bar{Y}\bar{Z}+\bar{Z}$	$XY(\bar{X}Y\bar{Z}+X\bar{Y}\bar{Z}+\bar{Z})$	$XY\bar{Z}$
0	0	0	1	1	1	0	0	0	1	0	0
0	0	1	1	1	0	0	0	0	0	0	0
0	1	0	1	0	1	0	1	0	1	0	0
0	1	1	1	0	0	0	0	0	0	0	0
1	0	0	0	1	1	0	0	1	1	0	0
1	0	1	0	1	0	0	0	0	0	0	0
1	1	0	0	0	1	1	0	0	1	1	1
1	1	1	0	0	0	1	0	0	0	0	0

From the above table we see that **columns 11 & 12 are identical** and hence the relation is **proved.**

The above example involves a proof involving three variables. Thus the columns 1, 2, and 3 give all the possible combinations of the three variables X, Y, and Z. We can follow a simple method to write the combinations properly.

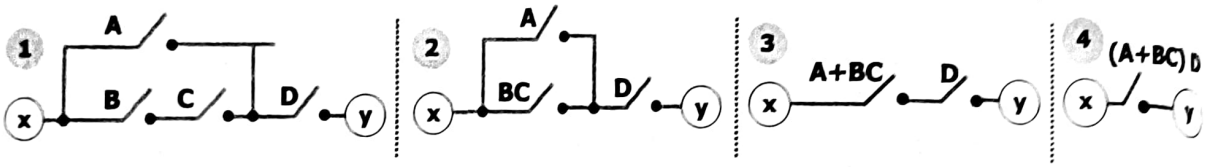
In general if we have 'n' number of variables, then the total number of combinations will be 2^n. In our case, n=3. Therefore we have a total of 2^3=8 combinations. Of these, the first column will have (8/2)=**4** zeroes followed by **4** ones. The second column will have (8/4)=**2** zeroes followed by **2** ones. The third column will have (8/8)=**1** zero followed by **1** one. Therefore if we have 4 variables (i.e. n=4) then the total combinations will be 2^4=16, of which the first column will have (16/2)=8 zeroes followed by 8 ones and so forth.

Switching Circuit Simplification

- Switching Circuit Simplification

The following examples are based on switching circuits that we had discussed in section 4.3, that deal with combination of switches in series or in parallel. The examples show how the final logical output of such a circuit can be derived by simplifying the circuit operation in a stepwise manner.

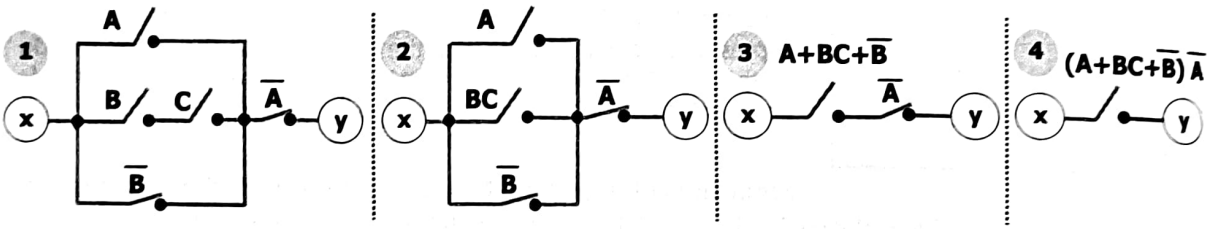
Example-14: Find the output of the following switching circuit between the terminals x and y:



A Parallel Circuit is equivalent to OR operation
A Series Circuit is equivalent to AND operation.

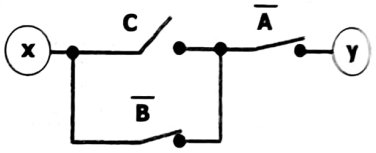
The method of finding the final output is shown in the diagrams above. From the first diagram we find that the switches **B and C are in series** and hence indicates an **AND operation**. The second diagram is a simplified version of the first with the two switches **B** and **C** being replaced by a single switch **BC** indicating the **AND** operation. Next we find that the switches **A and BC are in parallel** and hence indicates an **OR operation**. Therefore in the third figure, the switches have been replaced by a single switch **(A+BC)** indicating the **OR** operation between **A** and **BC**. Finally, the switches **(A+BC) and D being in series**, are replaced by an **AND operation** as shown in the final figure to derive at the final output as: **F = (A+BC) D**

Example-15: Simplify the circuit between the terminals x and y and derive at a simplified circuit.



In the above circuit, the terms $\bar{A}$ and $\bar{B}$ represent closed switches corresponding to the open switches A and B. The final output can be simplified to get the result:

$$
\begin{aligned}
F &= (A + BC + \bar{B})\,\bar{A} \\
&= (A + \bar{B} + C)\,\bar{A} \qquad \{\text{Since } X+\bar{X}Y = X+Y, \text{ and } X = \bar{B} \text{ here}\} \\
&= A\bar{A} + \bar{A}\bar{B} + C\bar{A} \\
&= \bar{A}\bar{B} + C\bar{A} \\
&= \bar{A}(\bar{B} + C)
\end{aligned}
$$



The simplified circuit is shown on the right. The parallel switches C and $\bar{B}$ indicate the OR operation ($\bar{B}$+C) and the series switch $\bar{A}$ indicates the AND operation $\bar{A}(\bar{B} + C)$ between $\bar{A}$ and $(\bar{B} + C)$.

De' Morgan's Theorems

4.5 De' Morgan's Theorems & Basic Duality of Boolean Rules

The expressions that we have worked with so far, had complemented variables like $\bar{A}$, $\bar{B}$, $\bar{C}$, etc. But there were no expressions were the complement was applied on an expression like $\overline{A(\bar{B} + C)}$, which involve a set of Boolean variables and operations. In case such an expression needs to be simplified, we need to remove the outer complement. Removing the outer complement can be done using a set of theorems called the De' Morgan's theorems.

De' Morgan's theorems give us the equivalent expressions for the complement of the logical sum or logical product of two Boolean variables or expressions. They state that:

- **Sum Theorem**: The complement of the logical sum of two variables X and Y is equal to the logical product of the individual complements of the variables.
  Thus $\overline{X + Y} = \overline{X} \cdot \overline{Y}$, in general, $\overline{W + X + Y + Z + \ldots} = \overline{W} \cdot \overline{X} \cdot \overline{Y} \cdot \overline{Z} \ldots$
- **Product Theorem**: The complement of the logical product of two variables X and Y is equal to the logical sum of the individual complements of the variables.
  Thus $\overline{X \cdot Y} = \overline{X} + \overline{Y}$, in general, $\overline{W \cdot X \cdot Y \cdot Z \cdot \ldots} = \overline{W} + \overline{X} + \overline{Y} + \overline{Z} + \ldots$

Statements of De' Morgan's Theorems

The **steps involved** in getting the right side of each expression are:

1. **Remove** the complement
2. **Replace** the '+' symbols with '·' symbols, and the '·' symbols with '+' symbols
3. **Complement** each variable of the expression

Sum Theorem: $X + Y = X \,\overline{\cdot}\, Y$

Product Theorem: $X \,\overline{\cdot}\, Y = X + Y$

To evaluate an expression that contains a combination of '+' (OR) and '·' (AND) operations, remember that the **AND operation has a higher priority over an OR operation**. This means that all AND operations in an expression are to be evaluated first, before evaluating any OR operation. If this rule is not followed, you will get an incorrect result. The following example will explain the point.

Example-16: Simplify the expression: $\overline{X\,Y + Z}$

The expression = $\overline{X\,Y} \cdot \overline{Z} = (\overline{X} + \overline{Y}) \cdot \overline{Z} = \overline{X}\,\overline{Z} + \overline{Y}\,\overline{Z}$

Remember, for a mixed Boolean expression: $\overline{X \cdot Y + Z} = \overline{X} \cdot \overline{Z} + \overline{Y} \cdot \overline{Z}$

In the above expression we have taken into account the priority of the AND operation over the OR operation while removing the complement for the **AND** term X Y in $\overline{X\,Y} \cdot \overline{Z}$. The term was placed within a pair of brackets to get $(\overline{X}+\overline{Y}) \cdot \overline{Z}$. If the brackets are not given then the result would be $\overline{X}+\overline{Y} \cdot \overline{Z}$, **which is wrong**.

- ## Proof of De Morgan's Theorems

The De Morgan's theorems can be proved using the **perfect induction method** or by using **algebraic method.** Let us first prove the theorems using the Perfect Induction method.

The truth table shown on the right is prepared taking into account all the terms involved in the above theorems. From the truth table, we find that **columns 6 and 7 are identical**, hence proving the first theorem and also columns **9 and 10 are identical**, thereby proving the second theorem.

1	2	3	4	5	6	7	8	9	10
X	Y	$\overline{X}$	$\overline{Y}$	X+Y	$\overline{X+Y}$	$\overline{X} \cdot \overline{Y}$	X·Y	$\overline{X \cdot Y}$	$\overline{X} + \overline{Y}$
0	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	0	0	1	1
1	0	0	1	1	0	0	0	1	1
1	1	0	0	1	0	0	1	0	0

Perfect Induction proof of De' Morgan's Theorems

To prove the theorems using the algebraic method, we will consider the two fundamental properties of any Boolean variable **A** i.e. $\mathbf{A + \overline{A} = 1}$ and $\mathbf{A \cdot \overline{A} = 0}$

Using the above fundamental properties we can write:

- $(X+Y) + \overline{(X+Y)} = 1$ and $(X+Y) \cdot \overline{(X+Y)} = 0$ ..... (i) [If we take A = (X+Y)]
- $(X \cdot Y) + \overline{(X \cdot Y)} = 1$ and $(X \cdot Y) \cdot \overline{(X \cdot Y)} = 0$ ..... (ii) [If we take A = (X.Y)]

Algebraic Method proof of De' Morgan's Theorems

If $\overline{(X+Y)} = \overline{X} \cdot \overline{Y}$, and $\overline{(X \cdot Y)} = \overline{X} + \overline{Y}$, as stated by De Morgan's theorems are true, then we can replace $\overline{(X+Y)}$ by $\overline{X} \cdot \overline{Y}$ in (i) and $\overline{(X \cdot Y)}$ by $\overline{X} + \overline{Y}$ in (ii) and still get the same results for the fundamental properties.

$\therefore$ to prove the first theorem we need to show $(X+Y)+\overline{X} \cdot \overline{Y}=1$ and $(X+Y) \cdot \overline{X} \cdot \overline{Y}=0$ [Replacing $\overline{(X+Y)}$ by $\overline{X} \cdot \overline{Y}$]

To show $(X+Y) + \overline{X} \cdot \overline{Y} = 1$

L.H.S. $= (X+Y) + \overline{X} \cdot \overline{Y}$
$= X + \overline{X} \cdot \overline{Y} + Y$ [As $A + \overline{A}B = A+B$]
$= X + \overline{Y} + Y$ [As $A + \overline{A} = 1$]
$= X + 1$ [As $A + 1 = 1$]
$= 1$ = R.H.S

To show $(X+Y) \cdot \overline{X} \cdot \overline{Y} = 0$

L.H.S. $= (X+Y) \cdot \overline{X} \cdot \overline{Y}$
$= X \cdot \overline{X} \cdot \overline{Y} + Y \cdot \overline{X} \cdot \overline{Y}$ [As $A \cdot \overline{A} = 0$]
$= 0 \cdot \overline{Y} + 0 \cdot \overline{X}$ [As $0 \cdot A = 0$]
$= 0 + 0$
$= 0$ = R.H.S

To prove the second theorem we need to show $(X\cdot Y)+(\bar{X}+\bar{Y})=1$ and $(X\cdot Y)\cdot(\bar{X}+\bar{Y})=0$ [Replacing $\overline{(X\cdot Y)}$ by $\bar{X}+\bar{Y}$]

To show $(X\cdot Y) + (\bar{X}+\bar{Y}) = 1$

L.H.S. $= (X\cdot Y) + (\bar{X}+\bar{Y})$
$= X\cdot Y + \bar{X}+\bar{Y}$
$= \bar{X} + XY + \bar{Y}$ [As $A+\bar{A}B = A+B$]
$= \bar{X} + Y + \bar{Y}$
$= \bar{X} + 1$ [As $A+\bar{A} = 1$]
$= 1 = R.H.S$ [As $A+1 = 1$]

To show $(X\cdot Y)\cdot(\bar{X}+\bar{Y}) = 0$

L.H.S. $= (X\cdot Y)\cdot(\bar{X}+\bar{Y})$
$= X\cdot Y\cdot\bar{X} + X\cdot Y\cdot\bar{Y}$
$= X\cdot\bar{X}\cdot Y + X\cdot Y\cdot\bar{Y}$ [As $A\cdot\bar{A} = 0$]
$= 0\cdot Y + X\cdot 0$ [As $0\cdot A = 0$]
$= 0 + 0$
$= 0 = R.H.S$

Principle of Duality

- **The Principle of Duality**

De Morgan's theorems indicate a **basic duality in any Boolean expression**. To put in another way, if a Boolean expression can be proved to be true then, its pair or dual obtained by complementing the expression and using the above theorems to simplify them, will also be true and need not be proved separately. This is because each AND operation has its equivalent OR operation and vice versa. These are equivalent by virtue of the above theorems. All Boolean expressions therefore can have more than one type of representation which are **equivalent statements** and proving one statement automatically establishes it's dual.

Duality Principle: *If a Boolean expression can be proved to be true then, its dual obtained by complementing the expression will also be true*

All the Boolean algebra rules show this duality i.e. the **product rules can be derived from the sum rules**. To make the idea clear let us take any two rules, apply the above theorem and find out what we get.

To find the dual of $X+1 = 1$

$X + 1 = 1$
$\therefore\ \overline{X+1} = \bar{1}$ (complementing both sides)
Or $\bar{X}\cdot\bar{1} = 0$ (applying De Morgan's theorem)
Or $Y\cdot 0 = 0$ (replacing $\bar{X}$ by Y)

The above result is same as the product rule $X\cdot 0=0$ and hence is the dual of $X+1=1$

To find the dual of $X+X = X$ (Idempotent Law)

$X + X = X$
$\therefore\ \overline{X+X} = \bar{X}$ (complementing both sides)
Or $\bar{X}\cdot\bar{X} = \bar{X}$ (applying De Morgan's theorem)
Or $Y\cdot Y = Y$ (replacing $\bar{X}$ by Y)

The above result shows that the rules $X+X=X$ and $X\cdot X=X$ are the duals of each other

There is a **simpler rule** of finding the dual of an expression. To find the dual, replace an AND operation with OR, and an OR operation with AND. Next replace a '0' by '1' and a '1' by '0' i.e. by their complement.

Therefore the dual of **X+1=1** will be **X·0=0** [by replacing OR by AND, and 1 by 0].

Similarly the dual of **X+X=X** will be simply **X·X=X** [by replacing the OR symbol by the AND symbol].

The dual of **(X+Y)·(X+Z)=X+Y·Z** will be $(X\cdot Y)+(X\cdot Z)=X\cdot(Y+Z)$ i.e. **X·Y+X·Z=X·(Y+Z)**

REMEMBER: While using this method keep in mind the priority of the AND operation and put all AND terms within brackets before you apply the rules. Otherwise you may get a wrong result as indicated in Example-16.

Example-17: Find the complement of the following Boolean expression: $\bar{X}\,Y\,\bar{Z} + X\,\bar{Y}\,\bar{Z} + \bar{Z}$

The complement of the expression $= \overline{\bar{X}\,Y\,\bar{Z} + X\,\bar{Y}\,\bar{Z} + \bar{Z}}$

$= \overline{(\bar{X}\,Y\,\bar{Z})}\cdot\overline{(X\,\bar{Y}\,\bar{Z})}\cdot\bar{\bar{Z}}$ *{Using De Morgan's Theorem}*

$= (\bar{\bar{X}} + \bar{Y} + \bar{\bar{Z}})\cdot(\bar{X} + \bar{\bar{Y}} + \bar{\bar{Z}})\cdot Z$ *{Using De Morgan's Theorem}*

$= (X + \bar{Y} + Z)\cdot(\bar{X} + Y + Z)\cdot Z$

$= [\,Z + (X + \bar{Y})]\cdot[\,Z + (\bar{X} + Y)\,]\cdot Z$

$= [\,Z + (X + \bar{Y})(\bar{X} + Y)\,]\cdot Z$ *{Using rule (A+B)(A+C)=A+BC}*

$= [\,Z + (X\,\bar{X} + X\,Y + \bar{Y}\,\bar{X} + \bar{Y}\,Y)\,]\cdot Z$

$= (\,Z + 0 + X\,Y + \bar{X}\,\bar{Y} + 0\,)\cdot Z$

$= Z\,Z + X\,Y\,Z + \bar{X}\,\bar{Y}\,Z$

$= Z + X\,Y\,Z + \bar{X}\,\bar{Y}\,Z$

$= Z\,(\,1 + X\,Y + \bar{X}\,\bar{Y}\,) = Z$ *{As (1+A) = 1}*

The above problem can also be simplified by simplifying the un-complemented expression first and then complementing the result as shown below:

The un-complemented expression $= \bar{X}\,Y\,\bar{Z} + X\,\bar{Y}\,\bar{Z} + \bar{Z}$

$= (\bar{X}\,Y + X\,\bar{Y})\,\bar{Z} + \bar{Z}$

$= [(\bar{X}\,Y + X\,\bar{Y}) + 1] \cdot \bar{Z} = \bar{Z}$

The complement of the expression $= \bar{\bar{Z}} = Z$ *{Same result as above}*

Example-18: Find the complement of the following Boolean expression: $\bar{X}\,Y + X\,\bar{Y}$

The complement of the expression $= \overline{\bar{X}\,Y + X\,\bar{Y}}$

$= (\overline{\bar{X}\,Y}) \cdot (\overline{X\,\bar{Y}})$

$= (\bar{\bar{X}} + \bar{Y}) \cdot (\bar{X} + \bar{\bar{Y}})$

$= (X + \bar{Y}) \cdot (\bar{X} + Y)$

$= (X\,\bar{X} + X\,Y + \bar{Y}\,\bar{X} + \bar{Y}\,Y)$

$= 0 + X\,Y + \bar{Y}\,\bar{X} + 0$

$= X\,Y + \bar{X}\,\bar{Y}$

Example-19: Simplify the following expression: $\overline{\overline{\overline{A\,B}\,C}\,D}$

The expression $= \overline{\overline{\overline{A\,B}\,C} + \bar{D}}$ *{removing the outer bar using De Morgan's theorem}*

$= \overline{A\,B}\,C + \bar{D}$ *{since $\bar{\bar{X}} = X$}*

$= (\bar{A} + \bar{B})\,C + \bar{D}$ *{applying De Morgan's theorem}*

$= \bar{A}\,C + \bar{B}\,C + \bar{D}$ *{removing the brackets}*

Example-20: Simplify the following expression: $(\overline{X+Y} + Y)\,(\overline{\bar{X}+Y})$

The expression $= (\bar{X} \cdot \bar{Y} + Y)\,(\bar{\bar{X}} \cdot \bar{Y})$

$= (\bar{X} \cdot \bar{Y} + Y)\,(X \cdot \bar{Y})$

$= \bar{X}\,\bar{Y}\,X\,\bar{Y} + Y\,X\,\bar{Y}$

$= 0 + 0$

$= 0$

Example-21: Simplify the following expression: $\overline{(\bar{X} + Y)} \cdot (\overline{\bar{Y}\,Z + \bar{X}})$

The expression $= (\bar{\bar{X}} \cdot \bar{Y}) \cdot [\bar{\bar{Y}} \cdot Z \cdot \bar{\bar{X}}]$

$= X\,\bar{Y}\,[(\bar{\bar{Y}} + \bar{Z}) \cdot X]$

$= X\,\bar{Y}\,[(Y + \bar{Z})\,X]$

$= X\,\bar{Y}\,(Y\,X + \bar{Z}\,X)$

$= X\,\bar{Y}\,Y\,X + X\,\bar{Y}\,\bar{Z}\,X$

$= 0 + X\,\bar{Y}\,\bar{Z}$

$= X\,\bar{Y}\,\bar{Z}$

4.6 SOP, POS, Min Term & Max Term Expressions

SOP and POS expressions

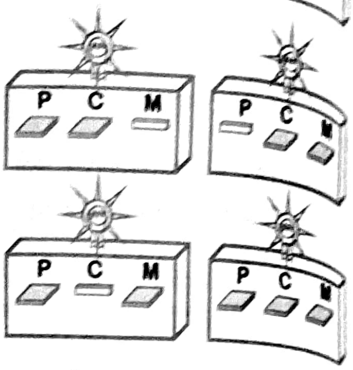
To design a logic circuit we have to **first derive at a logical mathematical expression** from which the circuit is designed. The logical expression is derived from the available inputs and the nature of the problem. Based on the type of the problem, a **truth table is constructed** with the available inputs. From the truth table using proper techniques that we will learn now, the desired expression is derived. An example will make the idea clear and help us learn the technique used.

Let us consider the **process of admission in a college for students**. A particular college has set up the criteria that a student has to score a minimum of 75% in any two of the subjects consisting of **Physics**, **Chemistry**, and **Maths** to get qualified for admission.

The college has a **hypothetical machine** which has three buttons for input and a bulb which glows as an output. The picture of the machine is shown to the right. An 'ON' switch corresponds to the fact that a student has secured a minimum of 75% marks in that subject and an 'OFF' switch corresponds to the fact that the student has got less than 75% marks.



The different possibilities under which the bulb can glow are shown on the right. The bulb glows if a student gets 75% in Physics and Chemistry, **OR** gets 75% in Chemistry and Maths, **OR** gets 75% in Physics and Maths, **OR** gets 75% in all the subjects. For all **other conditions, the bulb will not glow.**



Our aim is to design a circuit which when placed inside the box and connected to the switches, will **make the bulb glow as per the above conditions**. For this we take 4 Boolean variables **P, C, M** and **A** and construct a truth table satisfying the above condition, such that:

P = 0 indicates marks in Physics <75%
C = 0 indicates marks in Chemistry <75%
M = 0 indicates marks in Maths <75%
A = 0 indicates **not satisfied any criteria**

P = 1 indicates marks in Physics >=75%
C = 1 indicates marks in Chemistry >=75%
M = 1 indicates marks in Maths >=75%
A = 1 indicates **satisfied any one of the criteria**

A min term for a given row is formed by making the logical product of the input variables equal to 1.

In the truth table shown on the right, columns 1, 2 and 3 indicate the different combinations of marks that a student can get. **Column 4** represents the condition that a student has qualified for the **admission criteria A.** Whenever the **condition is satisfied, A is assigned a value of 1.**

As per the problem, the condition is satisfied for [P,C,M] = [0,1,1], [1,0,1], [1,1,0], & [1,1,1] (indicated by the rows 4, 6, 7, and 8).

1	2	3	4	5	6	7
P	C	M	A	Condition Satisfied	Product / Min Terms	Min term Number
0	0	0	0	No	$\bar{P}\,\bar{C}\,\bar{M}$	m_0
0	0	1	0	No	$\bar{P}\,\bar{C}\,M$	m_1
0	1	0	0	No	$\bar{P}\,C\,\bar{M}$	m_2
0	1	1	1	Yes	$\bar{P}\,C\,M$	m_3
1	0	0	0	No	$P\,\bar{C}\,\bar{M}$	m_4
1	0	1	1	Yes	$P\,\bar{C}\,M$	m_5
1	1	0	1	Yes	$P\,C\,\bar{M}$	m_6
1	1	1	1	Yes	$P\,C\,M$	m_7



Creating the min-term column

Column 6 is the '**min-term**' column. To construct an SOP expression:

- First type the product of the variables (P·C·M) in all cells in the min-term column
- Next **complement the variables which have a value '0'** in the P, C, M, columns corresponding to that row.

  For example in the third row [P,C,M] = [0,1,0]. Accordingly, since P=M=0, we put a bar over P and M respectively to make them 1 and get the min term as $\bar{P}\,C\,\bar{M}$. Similarly for the sixth row [P,C,M] = [1,0,1] and hence the min term is $P\,\bar{C}\,M$, as only the variable C has a value of '0'. Similarly we fill the rest of the cells in column 6.

  The **aim is to make the product of the variables in each min-term equal to 1** for that row.
- Finally select **only those min-terms** which have a **value of '1' in the output column 'A'** (indicated by the shaded cells in column 6) and **add them** to get the final SOP result.

Hence in this case the final result will be:

$$A = \bar{P}\,C\,M + P\,\bar{C}\,M + P\,C\,\bar{M} + P\,C\,M$$

An SOP expression is formed by logically adding the min-terms for rows with output as 1.

This is the required expression which satisfies all the conditions of our problem. To test its validity, simply put a combination of PCM in the expression and find what the output becomes.

For [P,C,M] = [1,0,0] we have **A**

$$= \bar{P}\,C\,M + P\,\bar{C}\,M + P\,C\,\bar{M} + P\,C\,M$$
$$= \bar{1}\,0\,0 + 1\,\bar{0}\,0 + 1\,0\,\bar{0} + 1\,0\,0$$
$$= 0{\cdot}0{\cdot}0 + 1{\cdot}1{\cdot}0 + 1{\cdot}0{\cdot}1 + 1{\cdot}0{\cdot}0 \quad \{as\ \bar{1}=0\ and\ \bar{0}=1\}$$
$$= 0 + 0 + 0 + 0$$
$$= 0$$

as is evident from row 5 of the truth table.

For [P,C,M] = [1,0,1] we have **A** $= \bar{P}\,C\,M + P\,\bar{C}\,M + P\,C\,\bar{M} + P\,C\,M$

$= \bar{1}\,0\,1 + 1\,\bar{0}\,1 + 1\,0\,\bar{1} + 1\,0\,1$

$= 0{\cdot}0{\cdot}1 + 1{\cdot}1{\cdot}1 + 1{\cdot}0{\cdot}0 + 1{\cdot}0{\cdot}1$

$= 0 + 1 + 0 + 0$

$= 1$ as is evident from row 6 of the truth table.

The above result A is known as the Sum Of Products or SOP form of output. The general rule to form an SOP expression is as follows: Wherever the desired output is 1, take the min-terms for those outputs and logically add those min-terms to get the final result.

Rule for SOP

Each min-term can also be represented by a **compact notation** as shown in **column 7** of the above table, with the letter '**m**' followed by an index. Note that the index starts from the value '0' and goes up to '7' for the eight min-terms. The **index value basically corresponds to the decimal equivalent of the binary number** formed by the variables. For example, for [P,C,M] = [1,0,1], the min-term is m_5, as the binary value $(101)_2 = (5)_{10}$.

Using the above notation, the expression **A** can be written as:

A $= \bar{P}\,C\,M + P\,\bar{C}\,M + P\,C\,\bar{M} + P\,C\,M$

$= m_3 + m_5 + m_6 + m_7$

$= \Sigma(3, 5, 6, 7)$ where the symbol Σ is used to **indicate a summation operation**.

Σ notation for SOP expression

Example-22: Find the SOP expression for the logic given by the following truth table:

X	Y	Z	R	min terms
0	0	0	1	$\bar{X}\bar{Y}\bar{Z}$
0	0	1	0	
0	1	0	0	
0	1	1	1	$\bar{X}YZ$
1	0	0	1	$X\bar{Y}\bar{Z}$
1	0	1	0	
1	1	0	0	
1	1	1	1	XYZ

Suppose for a particular problem we get a truth table as shown in the figure on the left, where X, Y, Z are three variables and R is the output column.

To get the Boolean expression for the result 'R', we simply write the min-terms for those combinations where the value of R is '1'. The other terms need not be written as those will not be used in the final expression. The required output is given by:

R $= \bar{X}\bar{Y}\bar{Z} + \bar{X}YZ + X\bar{Y}\bar{Z} + XYZ$

$= m_0 + m_3 + m_4 + m_7 = \Sigma(0, 3, 4, 7)$

An SOP expression can be expressed in a compact sigma notation as:
$F = \Sigma m_j$ (j=0,1,...)

If required, one can also **simplify the above result** and get:

R $= \bar{X}\bar{Y}\bar{Z} + \bar{X}YZ + X\bar{Y}\bar{Z} + XYZ$

$= \bar{X}\bar{Y}\bar{Z} + X\bar{Y}\bar{Z} + \bar{X}YZ + XYZ = (\bar{X} + X)\,\bar{Y}\bar{Z} + (\bar{X} + X)\,YZ = \bar{Y}\bar{Z} + YZ$

There is **another method** for forming a **Boolean expression** satisfying a logic table, known as the **Product Of Sums** or **POS** method. The method is explained below with the help of an example.

POS expressions

Let us consider the previous problem of admission to a college based on the marks in Physics, Chemistry, and Maths.

1	2	3	4	5	6	7
P	C	M	A	Condition Satisfied	Sum/Max Terms	Max term Number
0	0	0	0	**No**	P+C+M	M_0
0	0	1	0	**No**	P+C+$\bar{M}$	M_1
0	1	0	0	**No**	P+$\bar{C}$+M	M_2
0	1	1	**1**	Yes	P+$\bar{C}$+$\bar{M}$	M_3
1	0	0	0	**No**	$\bar{P}$+C+M	M_4
1	0	1	**1**	Yes	$\bar{P}$+C+$\bar{M}$	M_5
1	1	0	**1**	Yes	$\bar{P}$+$\bar{C}$+M	M_6
1	1	1	**1**	Yes	$\bar{P}$+$\bar{C}$+$\bar{M}$	M_7

In finding the POS expression **Column 6** now becomes the '**max-term**' column. To construct this column, first type the **sum of the variables** P+C+M in all cells. Next **put a bar over the variables which have a value '1'** in the P C M column corresponding to that row. For example [P,C,M] = [0,1,0] in the third row. Accordingly, since C=1, we put a bar over C to make it 0 and make the Max term as P+$\bar{C}$+M i.e. 0. Similarly for the sixth row [P,C,M] = [1,0,1] and hence the Max term is $\bar{P}$+C+$\bar{M}$, as the variables P and M have a value of '1'. In this way we construct the rest of the cells in column 6. We are basically **making the sum of the variables in each max-term equal to 0** for that row.

A max term for a given row is formed by making the logical sum of the input variables equal to 0.

Finally select **only those max-terms** which have a **value of '0' in the output column 'A'** (indicated by the coloured highlighted cells in column 6) and **multiply them** to get the final result. Hence in this case the final result will be:

A = (P+C+M) (P+C+$\bar{M}$) (P+$\bar{C}$+M) ($\bar{P}$+C+M)

A POS expression is formed by logically multiplying the max-terms for rows with output as 0.

This is the required expression which satisfies all the conditions of our problem. To test its validity, simply put a combination of P,C,M in the expression and find what the output becomes.

For [P,C,M] = [1,0,0] we have **A** = (P+C+M) (P+C+$\bar{M}$) (P+$\bar{C}$+M) ($\bar{P}$+C+M)
= (1+0+0) (1+0+$\bar{0}$) (1+$\bar{0}$+0) ($\bar{1}$+0+0)
= (1+0+0) (1+0+1) (1+1+0) (0+0+0) *{as $\bar{1}$=0 and $\bar{0}$=1}*
= 1 . 1 . 1 . 0
= 0 as is evident from row 5 of the truth table.

For [P,C,M] = [1,0,1] we have **A** = (P+C+M) (P+C+$\bar{M}$) (P+$\bar{C}$+M) ($\bar{P}$+C+M)
= (1+0+1) (1+0+$\bar{1}$) (1+$\bar{0}$+1) ($\bar{1}$+0+1)
= (1+0+1) (1+0+0) (1+1+1) (0+0+1) *{as $\bar{1}$=0 and $\bar{0}$=1}*
= 1 . 1 . 1 . 1
= 1 as is evident from row 6 of the truth table.

Rule for POS

The above result P is known as the Product Of Sums or POS form of output. The general rule to form a POS expression is as follows: Wherever the desired output is 0, take the Max-terms for those outputs and multiply those Max-terms to get the final result.

Each **max-term** can also be represented by a **compact notation** as shown in **column 7** of the above table, with the letter **'M'** followed by an index. Note that the index starts from the value '0' and goes up to '7' for the eight Max-terms. Using the above notation, the expression 'A' can be written as:

A = (P+C+M) (P+C+$\bar{M}$) (P+$\bar{C}$+M) ($\bar{P}$+C+M)
= $M_0 \cdot M_1 \cdot M_2 \cdot M_4$
= Π (0, 1, 2, 4) where the symbol Π is used to **indicate a product operation**.

An POS expression can be expressed in a compact pi notation as:
$F = \Pi M_j (j=0,1,...)$

Example-23: Let us find out the desired expression for the condition that the output in a circuit will be '1' whenever the **binary combination of its 3 inputs** gives a **prime number**.

1 Dec. No.	2 A	3 B	4 C	5 P	6 Condition Satisfied	7 Sum/Max Terms	8 Min term Number
0	0	0	0	**0**	**No**	A+B+C	M_0
1	0	0	1	**0**	**No**	A+B+$\bar{C}$	M_1
2	0	1	0	1	Yes		M_2
3	0	1	1	1	Yes		M_3
4	1	0	0	**0**	**No**	$\bar{A}$+B+C	M_4
5	1	0	1	1	Yes		M_5
6	1	1	0	**0**	**No**	$\bar{A}$+$\bar{B}$+C	M_6
7	1	1	1	1	Yes		M_7

The truth table for the above logic function realisation is shown on the right. **Column 1 gives the decimal values** of the corresponding binary number in columns 2, 3, and 4. For example the decimal number 3 is written beside the binary combination $(011)_2 = (3)_{10}$.

The range of values possible using 3 binary digits is 0 to 7 as shown in Column 1. The **prime numbers within this range are 2, 3, 5, and 7.** Accordingly output column 'P' has values of '1' corresponding to these values only.

Column 7 is the **max-term column.** We have taken only those max terms for which the output value is 0. Finally we have selected only those Max terms and multiplied them to get the final POS result as:

P = (A+B+C) (A+B+$\bar{C}$) ($\bar{A}$+B+C) ($\bar{A}$+$\bar{B}$+C)

This is the required expression which satisfies all the conditions of our problem. The compact notation for this expression will be:

$P = (A+B+C)\,(A+B+\bar{C})\,(\bar{A}+B+C)\,(\bar{A}+\bar{B}+C)$
$= M_0 \cdot M_1 \cdot M_4 \cdot M_6$
$= \Pi\,(0, 1, 4, 6)$

Example-24: Find the POS expression for the logic given by the following truth table:

A	B	C	R	Max terms
0	0	0	**1**	
0	0	1	**0**	$A+B+\bar{C}$
0	1	0	**1**	
0	1	1	**0**	$A+\bar{B}+\bar{C}$
1	0	0	**1**	
1	0	1	**0**	$\bar{A}+B+\bar{C}$
1	1	0	**1**	
1	1	1	**1**	

Suppose for a particular problem we get a truth table as shown in the figure on the left, where A, B, C are the three variables and R is the output column.

To get the Boolean expression for the result 'R', **we simply write the max-terms** for those combinations where the value of R is '0'. The others need not be written as those will not be used in the final expression. The required output is given by:

$R = (A+B+\bar{C})\,(A+\bar{B}+\bar{C})\,(\bar{A}+B+\bar{C})$
$= M_1 \cdot M_3 \cdot M_5 = \Pi\,(1, 3, 5)$

If required, one can also simplify the above result.

4.7 Canonical forms of Boolean Expressions and their Complements

Canonical form of Boolean expressions

When a Boolean expression in the SOP form has **product terms** each of which **contains all the variables** (either in complemented or in non-complemented form) then the expression is called a **Canonical SOP**. Similarly if a Boolean expression in the POS form contain sum terms each containing all the variables (either in complemented or in non-complemented form) then the expression is called a **Canonical POS**. In short a Boolean **expression containing only full min terms or full max terms is called a standard or canonical form of expression**. The following examples will make the nomenclatures clear for 3 variable problems.

- $R_1 = \bar{X}\bar{Y}\bar{Z} + \bar{X}YZ + X\bar{Y}\bar{Z} + XY\bar{Z}$ → Canonical SOP as each term contains all the variables X,Y,Z
- $R_2 = YZ + \bar{X}\bar{Z} + \bar{X}YZ$ → Non-Canonical SOP as X,Y,Z are not present in all the terms
- $R_3 = (A+B+C)\,(A+B+\bar{C})\,(A+\bar{B}+\bar{C})$ → Canonical POS as each term contains all the variables A,B,C
- $R_4 = (A+C)\,(A+B+\bar{C})\,(B+\bar{C})$ → Non-Canonical POS as A,B,C are not present in all the terms
- $R_5 = AB + (A+\bar{B}+\bar{C})\,(B+\bar{C})$ → Non-Canonical expression as there are mixed terms

The importance of Canonical forms lies in the fact that a Canonical SOP or POS Boolean expression can also be written in a more **compact form** using the Σ or Π notations as we have seen in the last section.

Thus the expressions R_1 and R_3 can be written using the compact notation as:

$R_1 = \bar{X}\bar{Y}\bar{Z} + \bar{X}YZ + X\bar{Y}\bar{Z} + XY\bar{Z} = m_0 + m_3 + m_4 + m_6 = \Sigma(0, 3, 4, 6)$

$R_3 = (A+B+C)\,(A+B+\bar{C})\,(A+\bar{B}+\bar{C}) = M_0 \cdot M_1 \cdot M_3 = \Pi\,(0, 1, 3)$

Being non-Canonical in nature, this type of notation is not possible for the expressions R_2, R_4 and R_5.

The complement of canonical expressions can also be **obtained in a simple manner** (without using De Morgan's theorems).

Consider the truth table shown on the right. By observing the min and Max term columns you will find that each Max-term is the **complement** of the corresponding min-term **in the same row**. For example:

$\overline{m_1} = \overline{\bar{A}\,\bar{B}\,C} = \bar{A}+\bar{B}+\bar{C} = A+B+\bar{C} = M_1$

Using the min-terms the output R for the above truth table can be written as:

A	B	C	R	min terms	m_i	Max terms	M_i
0	0	0	**1**	$\bar{A}\,\bar{B}\,\bar{C}$	m_0	A+B+C	
0	0	1	**0**	$\bar{A}\,\bar{B}\,C$		$A+B+\bar{C}$	M_1
0	1	0	**0**	$\bar{A}\,B\,\bar{C}$		$A+\bar{B}+C$	M_2
0	1	1	**1**	$\bar{A}\,B\,C$	m_3	$A+\bar{B}+\bar{C}$	
1	0	0	**1**	$A\,\bar{B}\,\bar{C}$	m_4	$\bar{A}+B+C$	
1	0	1	**0**	$A\,\bar{B}\,C$		$\bar{A}+B+\bar{C}$	M_5
1	1	0	**0**	$A\,B\,\bar{C}$		$\bar{A}+\bar{B}+C$	M_6
1	1	1	**1**	$A\,B\,C$	m_7	$\bar{A}+\bar{B}+\bar{C}$	

$\mathbf{R} = \bar{A}\,\bar{B}\,\bar{C} + \bar{A}\,B\,C + A\,\bar{B}\,\bar{C} + A\,B\,C = m_0 + m_3 + m_4 + m_7 = \Sigma\,(0, 3, 4, 7)$

Therefore $\bar{\mathbf{R}}$ $= \overline{\bar{A}\,\bar{B}\,\bar{C} + \bar{A}\,B\,C + A\,\bar{B}\,\bar{C} + A\,B\,C}$

$= \overline{\bar{A}\,\bar{B}\,\bar{C}} \cdot \overline{\bar{A}\,B\,C} \cdot \overline{A\,\bar{B}\,\bar{C}} \cdot \overline{A\,B\,C}$ *[using De Morgan's theorem]*

$= \overline{m_0} \cdot \overline{m_3} \cdot \overline{m_4} \cdot \overline{m_7}$

$= M_0 \cdot M_3 \cdot M_4 \cdot M_7$ *[as complement of a min term is a Max term]*

$= \Pi\,(0, 3, 4, 7)$

For a canonical Boolean expression

$\Sigma(m_j) = \Pi(M_j)$

$\Pi(M_j) = \Sigma(m_j)$

Therefore in general: $\overline{\Sigma(m_j)} = \Pi(M_j)$ and $\overline{\Pi(M_j)} = \Sigma(m_j)$

A canonical expression can also be **easily converted from its SOP to POS form and vice versa**. From the expression of **R** from the previous table, it can be seen that **R** can be written in 2 ways, viz.

$\mathbf{R} = \bar{A}\,\bar{B}\,\bar{C} + \bar{A}\,B\,C + A\,\bar{B}\,\bar{C} + A\,B\,C = m_0 + m_3 + m_4 + m_7 = \Sigma\,(0, 3, 4, 7)$ and

$\mathbf{R} = (A+B+\bar{C})\cdot(A+\bar{B}+C)\cdot(\bar{A}+B+\bar{C})\cdot(\bar{A}+\bar{B}+C) = M_1 \cdot M_2 \cdot M_5 \cdot M_6 = \Pi\,(1, 2, 5, 6)$

i.e. $\mathbf{R} = \Sigma\,(0, 3, 4, 7) = \Pi\,(1, 2, 5, 6)$

It can be easily seen from the above relation that out of the total index values from 0 to 7, the **max term indices contain those values that are not there in the min term indices** and vice versa. Hence a SOP expression can easily be converted to its POS form and vice versa by simply noting the index values. The following example will make the idea clear.

Example-25: Change the following canonical SOP expression to its POS form: $A\,B\,\bar{C} + A\,\bar{B}\,C + \bar{A}\,B\,C$

Expression **R** $= A\,B\,\bar{C} + A\,\bar{B}\,C + \bar{A}\,B\,C$

$= m_6 + m_5 + m_3$

$= \Sigma\,(3, 5, 6)$

$= \Pi\,(0, 1, 2, 4, 7)$ *{as these indices are not present in the SOP expression}*

$= M_0 \cdot M_1 \cdot M_2 \cdot M_4 \cdot M_7 = (A+B+C)(A+B+\bar{C})(A+\bar{B}+C)(\bar{A}+B+C)(\bar{A}+\bar{B}+\bar{C})$

Example-26: Change the following canonical POS expression to SOP form: $(\bar{A}+B+\bar{C})(A+\bar{B}+C)(A+B+\bar{C})$

Expression **R** $= (\bar{A}+B+\bar{C})(A+\bar{B}+C)(A+B+\bar{C})$

$= M_5 \cdot M_2 \cdot M_1 = \Pi\,(1, 2, 5)$

$= \Sigma\,(0, 3, 4, 6, 7)$ *{as these indices are not present in the POS expression}*

$= m_0 + m_3 + m_4 + m_6 + m_7 = \bar{A}\,\bar{B}\,\bar{C} + \bar{A}\,B\,C + A\,\bar{B}\,\bar{C} + A\,B\,\bar{C} + A\,B\,C$

Converting from Σ or Π form to Canonical form

- **Converting from Σ or Π notation to Canonical Form**

In the **absence of a truth table** the min and max terms can be calculated directly from the Σ or Π notation as shown below. Remember that for a **3 variable problem** (e.g. A,B,C) the min term index will be from 0 to 7, and for a **4 variable problem** (e.g. A,B,C,D) the min term index will be from 0 to 15.

First get the binary equivalent of a min term index present in the Σ notation. To do this **represent each decimal digit index as a combination of 4, 2, and 1** for a three variable problem. Similarly for a four variable problem, represent each decimal index digit as a combination of 8, 4, 2, and 1.

Next put a '1' in place of the above digits (4,2,1 or 8,4,2,1) that are present in the combination and '0' in place of the above digits those are not present in the combination. The chart below will clarify the point.

		4	**2**	**1**
0 =	0+0+0 →	0	0	0
1 =	0+0+1 →	0	0	1
2 =	0+2+0 →	0	1	0
3 =	0+2+1 →	0	1	1
4 =	4+0+0 →	1	0	0
5 =	4+0+1 →	1	0	1
6 =	4+2+0 →	1	1	0
7 =	4+2+1 →	1	1	1

For m_4 we have the index as 4, which is written as 4+0+0 → 1 0 0 (as 4 is present and 2, 1 are absent)

We know that a min term is forcibly made equal to 1. Thus for the combination [1 0 0] convert the two 0's to 1 by taking the **complement of those values**. Therefore for a **three variable logical problem** the corresponding min term for m_4 will be $A\,\bar{B}\,\bar{C}$

Similarly for the min term $\mathbf{m_6}$, we have the index 6, which can be written as 4+0+2 → 1 0 1

Therefore the **3 variable** min term corresponding to m_6 will be $A\,\bar{B}\,C$

Note that we have put a bar over those variables which correspond to a 0.

For a **4 variable** min term m_9, the index 9 can be written as 8+0+0+1 → 1 0 0 1

Therefore the **4 variable** min term corresponding to m_9 will be $A\,\bar{B}\,\bar{C}\,D$

Next **to calculate the max term from the Π notation** we will use a similar process to get the binary equivalent of the max term index. After that we have to **forcibly make the sum of the variables equal to 0.** The following examples will clarify the process.

For $\mathbf{M_4}$ we have the index as 4, which is written as 4+0+0 → 1 0 0 (as 4 is present and 2, 1 are absent)

Thus for the combination [1 0 0] convert the 1 to 0 by taking the **complement of that value** to make the sum equal to 0. Therefore for a **three variable logical problem** the corresponding max term for M_4 will be $\bar{A}+B+C$

For a **4 variable** max term M_9, the index 9 can be written as 8+0+0+1 → 1 0 0 1

Therefore the **4 variable** max term corresponding to M_9 will be $\bar{A}+B+C+\bar{D}$

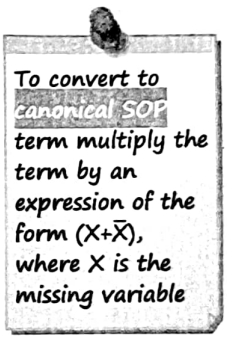
- Converting non-canonical SOP expression to its canonical form

A non-canonical expression can be converted to a canonical form also. **To convert a non-canonical SOP to its canonical form the following steps are to be followed**:

1. Multiply the non-canonical term with an expression of the form $(X+\bar{X})$ [since its value is equal to 1, the term remains unchanged after multiplication].
2. The multiplier is chosen in such a manner that it contains **that** variable which is **not** present in the original term.
3. The process is repeated until each term is converted to its canonical form.



Canonical SOP form of Boolean expressions



Example-27: Convert to its canonical form the expression: $f(X,Y) = \bar{X} + \bar{Y}$

This is a function with 2 variables. Therefore in the first term Y is missing and in the second term X is missing. Hence we have to multiply the first term by $(Y+\bar{Y})$ and the second term by $(X+\bar{X})$

$$\begin{aligned}\text{Expression} &= \bar{X}(Y+\bar{Y}) + \bar{Y}(X+\bar{X}) \\ &= \bar{X}Y + \bar{X}\bar{Y} + \bar{Y}X + \bar{Y}\bar{X} \\ &= \bar{X}Y + \bar{X}\bar{Y} + X\bar{Y} + \bar{X}\bar{Y} && \{\textit{rearranging the terms}\} \\ &= \bar{X}Y + \bar{X}\bar{Y} + X\bar{Y} && \{\textit{eliminating repeating terms since } A+A=A\} \\ &= m_1 + m_0 + m_2 \\ &= \Sigma(0, 1, 2)\end{aligned}$$

Example-28: Convert to its canonical form the expression: $f(X,Y,Z) = X\bar{Y} + \bar{Z} + \bar{X}Y\bar{Z}$

In the first term we find that Z is missing. Therefore we have to multiply the first term by $(Z+\bar{Z})$. Similarly in the second term both X and Y are missing. Therefore we have to multiply the second term by $(X+\bar{X})$ and $(Y+\bar{Y})$ one by one.

$$\begin{aligned}\text{Expression} &= X\bar{Y} + \bar{Z} + \bar{X}Y\bar{Z} \\ &= X\bar{Y}(Z+\bar{Z}) + \bar{Z}(X+\bar{X})(Y+\bar{Y}) + \bar{X}Y\bar{Z} \\ &= (X\bar{Y}Z + X\bar{Y}\bar{Z}) + \bar{Z}(XY + X\bar{Y} + \bar{X}Y + \bar{X}\bar{Y}) + \bar{X}Y\bar{Z}\end{aligned}$$

$= X\bar{Y}Z + X\bar{Y}\bar{Z} + \bar{Z}XY + \bar{Z}X\bar{Y} + \bar{Z}\bar{X}Y + \bar{Z}\bar{X}\bar{Y} + \bar{X}Y\bar{Z}$

$= X\bar{Y}Z + X\bar{Y}\bar{Z} + XY\bar{Z} + X\bar{Y}\bar{Z} + \bar{X}Y\bar{Z} + \bar{X}\bar{Y}\bar{Z} + \bar{X}Y\bar{Z}$ *{rearranging the terms}*

$= X\bar{Y}Z + X\bar{Y}\bar{Z} + XY\bar{Z} + \bar{X}Y\bar{Z} + \bar{X}\bar{Y}\bar{Z}$ *{eliminating repeating terms since A+A=A}*

$= m_5 + m_4 + m_6 + m_2 + m_0$

$= \Sigma\,(0, 2, 4, 5, 6)$



Canonical POS form of Boolean expressions

- **Converting non-canonical POS expression to its canonical form**

To convert a non-canonical POS to its canonical form the following steps are to be followed.

1. Add to the non-canonical term an expression of the form $X\bar{X}$ (since its value is equal to 0, the expression remains unchanged after addition).
2. Apply the distributive rule X+YZ = (X+Y)(X+Z)
3. The process is repeated until each term is converted to its canonical form.

To convert to canonical POS term add to the term an expression of the form $(X\cdot\bar{X})$, where X is the missing variable

Example-29: Convert to its canonical form the expression: $f(X,Y) = \bar{X}\,\bar{Y}$

There are 2 terms in the POS expression above. The first is $\bar{X}$ and the second one is $\bar{Y}$. Therefore we find that the variable Y is missing from the first term and the variable X is missing from the second term. Therefore we need to add $Y\,\bar{Y}$ in the first term and $X\,\bar{X}$ in the second term.

Expression $= \bar{X}\,\bar{Y}$

$= (\bar{X} + Y\,\bar{Y})\,(\bar{Y} + X\,\bar{X})$

$= (\bar{X} + Y)(\bar{X} + \bar{Y})\,(\bar{Y} + X)(\bar{Y} + \bar{X})$ *{applying the rule A+BC = (A+B)(A+C)}*

$= (\bar{X} + Y)(\bar{X} + \bar{Y})\,(X + \bar{Y})(\bar{X}+\bar{Y})$ *{rearranging the terms}*

$= (\bar{X} + Y)\,(\bar{X} + \bar{Y})\,(X + \bar{Y})$ *{removing the repeating terms as A.A=A}*

$= M_2 \cdot M_3 \cdot M_1$

$= \Pi\,(1, 2, 3)$

Example-30: Convert to its canonical form the expression: $f(X,Y) = \bar{Z}\,(X+\bar{Y})(\bar{X}+Y+\bar{Z})$

Let us take each term separately and convert each to its canonical form.

Term: $\bar{Z}$ $= \bar{Z} + X\,\bar{X}$ *{adding the missing X variable}*

$= (\bar{Z} + X)(\bar{Z} + \bar{X})$ *{since A+BC = (A+B)(A+C)}*

$= (\bar{Z} + X + Y\,\bar{Y})\,(\bar{Z} + \bar{X} + Y\,\bar{Y})$ *{adding the missing Y variable in each term}*

$= [\,(\bar{Z} + X) + Y\,\bar{Y}\,]\,[\,(\bar{Z} + \bar{X}) + Y\,\bar{Y}\,]$

$= [\,(\bar{Z} + X) + Y\,]\,[\,(\bar{Z} + X) + \bar{Y}\,]\,[\,(\bar{Z} + \bar{X}) + Y\,]\,[\,(\bar{Z} + \bar{X}) + \bar{Y}\,]$

$= (\,\bar{Z} + X + Y\,)\,(\,\bar{Z} + X + \bar{Y}\,)\,(\,\bar{Z} + \bar{X} + Y\,)\,(\,\bar{Z} + \bar{X} + \bar{Y}\,)$

$= (\,X + Y + \bar{Z}\,)\,(\,X + \bar{Y} + \bar{Z}\,)\,(\,\bar{X} + Y + \bar{Z}\,)\,(\,\bar{X} + \bar{Y} + \bar{Z}\,)$ *{rearranging the terms}*

Note that for the expression $(\bar{Z} + X + Y\,\bar{Y})$ we have taken the term $(\bar{Z} + X)$ as A, Y as B and $\bar{Y}$ as C for the distributive law A+BC = (A+B)(A+C).

Term: $(X+\bar{Y})$ $= (X+\bar{Y}) + Z\,\bar{Z}$

$= [\,(X + \bar{Y}) + Z\,]\,[\,(X + \bar{Y}) + \bar{Z}\,]$

$= (\,X + \bar{Y} + Z\,)\,(\,X + \bar{Y} + \bar{Z}\,)$

Therefore the Canonical Expression is equal to the product of the above terms:

Expression $= (\,X+Y+\bar{Z}\,)\,(\,X+\bar{Y}+\bar{Z}\,)\,(\,\bar{X}+Y+\bar{Z}\,)\,(\,\bar{X}+\bar{Y}+\bar{Z}\,)\,(\,X+\bar{Y}+Z\,)\,(\,X+\bar{Y}+\bar{Z}\,)\,(\,\bar{X}+Y+\bar{Z}\,)$

$= (\,X+Y+\bar{Z}\,)\,(\,X+\bar{Y}+\bar{Z}\,)\,(\,\bar{X}+Y+\bar{Z}\,)\,(\,\bar{X}+\bar{Y}+\bar{Z}\,)\,(\,X+\bar{Y}+Z\,)$ *{removing repeating terms}*

$= M_1 \cdot M_3 \cdot M_5 \cdot M_7 \cdot M_2$

$= \Pi\,(1, 2, 3, 5, 7)$

4.8 Techniques for Simplification: Using Karnaugh Map

Simplification using K'map

When the number of variables is more than three then it usually becomes difficult to form or simplify the necessary SOP expression. In such cases other methods are employed to find the result. One such method is a **graphical one** and is called the **Karnaugh Map** or **K' Map** method. It uses the rule $A+\bar{A} = 1$, repeatedly to solve or simplify an expression.

- Karnaugh Map for expressions with 2 variables:

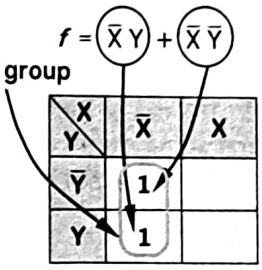
A K' map for SOP expressions use the rule $(X+\bar{X})=1$ repeatedly to simplify an expression

The right side figure shows the **Karnaugh Map representation** of min-terms for two variables X & Y. The different X values are written in the first row and the different Y values are written in the first column. Each cell at the **intersection of a row and a column** correspond to the **min-term** formed by the product of the X & Y terms in that row and column.

Y \ X	$\bar{X}$	X
$\bar{Y}$	$\bar{X}\bar{Y}$	$X\bar{Y}$
Y	$\bar{X}Y$	XY

For example if $f = \bar{X}Y + \bar{X}\bar{Y}$

On simplification we get $f = \bar{X}(Y + \bar{Y}) = \bar{X}$ {since $Y + \bar{Y} = 1$}



Rules to form a group in a K' map

To do this simplification using K'map, the following steps are to be followed:

1. Put a **'1' in cells** for min-terms which appear in the given SOP expression. See the K'map on the right for the expression $(\bar{X}Y + \bar{X}\bar{Y})$.
2. Next **group the 1's in adjacent cells** by taking as many number of 1's as possible within a group. A group is also called a **sub-cube**. Note that:
   - You may only **include either 8, 4, 2, or a single '1'** in a given group.
   - Group cells to **form a rectangular selection**. Selecting cells **diagonally is not allowed**.
   - You may need to form more than one group to include all the 1's. But do not form unnecessary groups. In general **include** the **maximum number of 1's** in a group and form the **minimum number of groups** until all the '1's are selected. Larger a group, more it can be simplified.
   - To form a group, first **look for the availability of 8 adjacent cells**. If not available, look for 4, then 2, and then 1, in that order. However **do not group 3, 5, 6, or 7 number of cells** even if these form adjacent cells.
   - To **increase the size of a given group**, you may include one or more '1's from another group. This is called **overlapping**. This does not change the expression as per the Boolean rule **A+A=A**

   In our example, as we have only two adjacent cells having 1's, we have circled them to form a group.
3. Each group will contribute to a term in the final output. To get the output of a given group write **only that term which remains unchanged within that group.**

   In our example we find that within the marked circle the terms present are $\bar{X}Y$ and $\bar{X}\bar{Y}$. In these terms $\bar{X}$ has remained same or unchanged in both the cells, while Y has changed from Y to $\bar{Y}$.
4. **Logically add all the terms** derived from each group to get the final expression.

   In this example, with a single group, the final output is given by: **f** = $\bar{X}$ (same result as above).

A rectangular group can contain either 8, 4, 2, or a single cell in a K' map

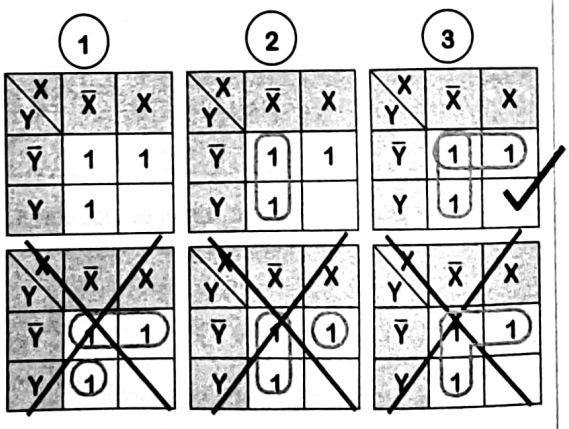
Example-31: Simplify $f = \bar{X}Y + \bar{X}\bar{Y} + X\bar{Y}$

In the K' Map we first put '1's in the squares containing the valid min-terms.

We then circle the pairs of adjacent squares containing 1's. Here we have only 3 number of 1's. Hence we cannot form a rectangle with 8 or 4 numbers of cells.

The next option is to use 2 cells. We make a vertical group with 2 cells as shown in figure-2 on the right.

To include the remaining '1' at the corner ($X\bar{Y}$ term), we pair it up with the '1' from the first group, as shown in figure-3. The overlapping is done to minimise the expression further. Without the overlapping it would have been a group with only a single cell in it.



The concept of overlapping can be confusing at first, but do not worry if you cannot understand now what this means, further examples will make the point clear.

In the horizontal group the **non-changing** term is $\bar{Y}$ as it is present in both the terms within the group and in the vertical group the **non-changing** term is $\bar{X}$ as it is present is both the terms within the vertical group. To get the final output we have to logically **add the terms derived from each group**.

Thus the final output is given by: $f = \bar{X} + \bar{Y}$

3-variable K' map

- Karnaugh Map for expressions with 3 variables:

The K' Map representation of min-terms with three variables is shown in the table on the right. In the header row, the different X, Y values and in the left column the different Z values are written. In writing the same, care has been taken so that **only a single variable changes in value between any two adjacent header cells**.

single variable change between cells

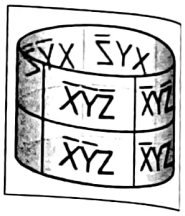
Z \ XY	$\bar{X}\bar{Y}$	$\bar{X}Y$	XY	$X\bar{Y}$
$\bar{Z}$	$\bar{X}\bar{Y}\bar{Z}$	$\bar{X}Y\bar{Z}$	$XY\bar{Z}$	$X\bar{Y}\bar{Z}$
Z	$\bar{X}\bar{Y}Z$	$\bar{X}YZ$	XYZ	$X\bar{Y}Z$

Remember there is always a single variable change between any two adjacent cells in a K' map

Hence in the third column, instead of writing $X\bar{Y}$ we have written XY. This can be further observed from an **alternate form of the first table** shown on the right. It is formed by using the binary input values of the variables X, Y, Z and their corresponding min-term numbers. Here the sequence of the header is (00, 01, 11, 10), instead of the usual binary sequence of (00, 01, 10, 11).

Z \ XY	00	01	11	10
0	m_0	m_2	m_6	m_4
1	m_1	m_3	m_7	m_5

The subscripts of the different terms (0, 1, 2, ... 7) indicate the order in which the min-terms are placed in the table corresponding to their increasing binary values. Note the change in values in the last two columns.



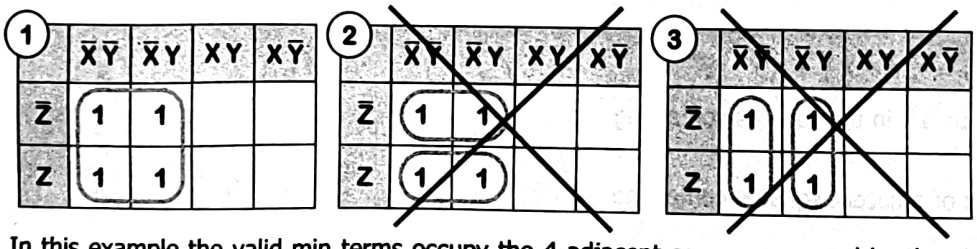
Furthermore it can be seen that the first and the last columns also differ by a single variable and hence the table can be thought to be rolled to form a cylinder with the right and left edges of the table joined. The figure on the right clarifies this point. Thus $\bar{X}\bar{Y}\bar{Z}$ and $X\bar{Y}\bar{Z}$ are actually adjacent terms and so are $\bar{X}\bar{Y}Z$ and $X\bar{Y}Z$.

Example-32: The problem is given by a three variable truth table shown on the right. It is seen that the output is high for the input values 0 to 3 only.

X	Y	Z	f
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

The min-terms corresponding to the output of the truth table are $\bar{X}\bar{Y}\bar{Z} + \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + \bar{X}YZ$. These are then plotted into the K' Map as shown below.

We know that we can group either 8, 4, 2, or 1 cell. But here we find that we cannot make a group with 8 cells as there are only 4 number of 1's. The next option is to take 4 cells and it is possible to do so in this case. Thus a single group is constructed encircling all the 1's. Since we should make the minimum number of groups, we have made a single group as shown in figure-1 instead of two groups (as shown in figures 2 and 3).



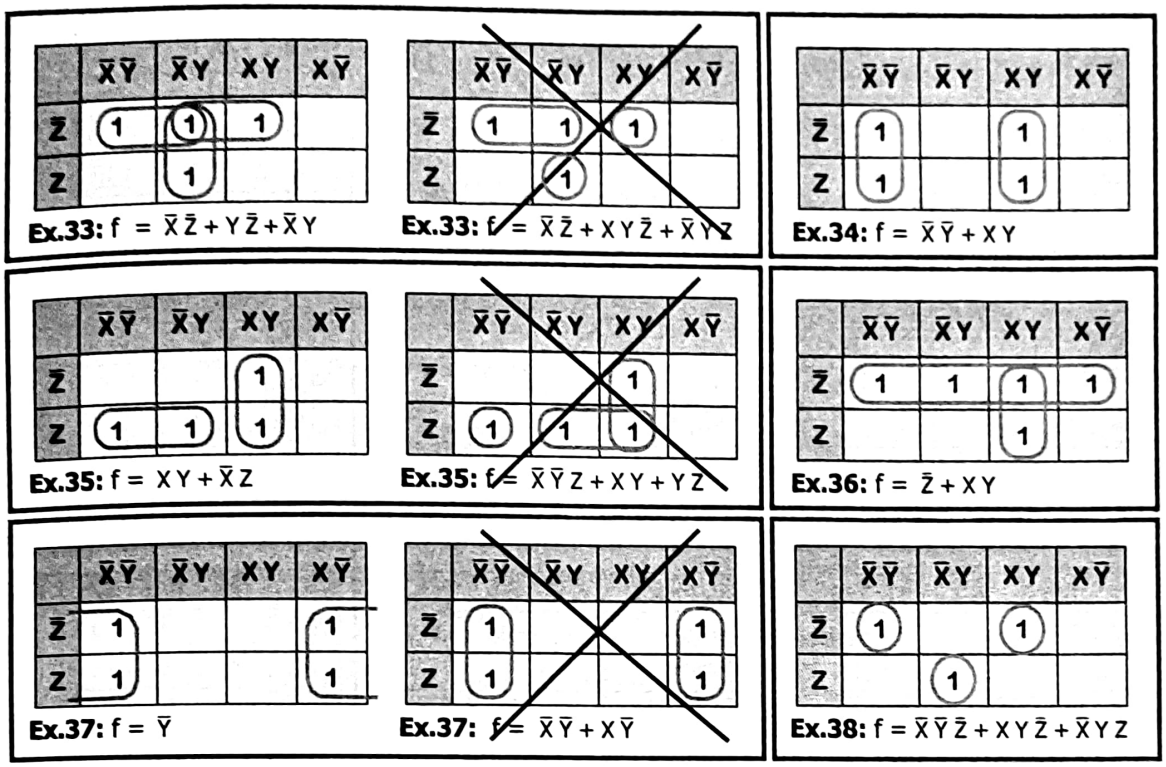
In this example the valid min terms occupy the 4 adjacent squares grouped by the circle. If we simplify the SOP expression algebraically we find:

$$\begin{aligned} f &= \bar{X}\bar{Y}\bar{Z} + \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + \bar{X}YZ \\ &= \bar{X}\bar{Z}(\bar{Y} + Y) + \bar{X}Z(\bar{Y} + Y) \\ &= \bar{X}\bar{Z} + \bar{X}Z \\ &= \bar{X}(\bar{Z} + Z) \\ &= \bar{X} \end{aligned}$$

To find out the simplified expression from the K' Map directly, note that within the four squares **the variable that remains unchanged within the group is** $\bar{X}$. The other variables i.e. Y and Z have changed in values within the group. As per rule the simplified term for the group is that value which remains unchanged.

Hence, for the single group we have **f** = $\bar{X}$ as has already been obtained algebraically.

Examples given below clarify the simplifying process. Note that each expression contains only those variables which remain unchanged within the group. Moreover the groups are made in such a manner that the maximum number of min-terms are grouped together to form a valid group.



Examples of 3 variable K' map

Note that in **example 37**, we have made a **single group** containing the four 1's, as the two **extreme columns** are also **adjacent to each other** as has been shown in the previous page.

- Karnaugh Map for 4 variables:

The K'map for min-terms with four variables is drawn with the first two variables in the header row and the second two in the leftmost column. Similar to a 3 variable map, the terms are written such that there is a **single variable difference between any two adjacent cells.** Here the map can be thought of being rolled both horizontally along its left and right margins and vertically along its top and bottom margins. Thus the **columns 1 & 4 are adjacent** and so are the **rows 1 & 4** (see figure above) and the number of cells possible in the largest group is equal to 8.

Adjacent rows / *Adjacent columns*

	$\bar{W}\bar{X}$	$\bar{W}X$	WX	$W\bar{X}$
$\bar{Y}\bar{Z}$	$\bar{W}\bar{X}\bar{Y}\bar{Z}$	$\bar{W}X\bar{Y}\bar{Z}$	$WX\bar{Y}\bar{Z}$	$W\bar{X}\bar{Y}\bar{Z}$
$\bar{Y}Z$	$\bar{W}\bar{X}\bar{Y}Z$	$\bar{W}X\bar{Y}Z$	$WX\bar{Y}Z$	$W\bar{X}\bar{Y}Z$
YZ	$\bar{W}\bar{X}YZ$	$\bar{W}XYZ$	$WXYZ$	$W\bar{X}YZ$
$Y\bar{Z}$	$\bar{W}\bar{X}Y\bar{Z}$	$\bar{W}XY\bar{Z}$	$WXY\bar{Z}$	$W\bar{X}Y\bar{Z}$

4 variable K' map

The figure on the right shows the distribution of terms of a four variable K'map in terms of the **binary input values of the variables WXYZ and their corresponding min-term** numbers. Remembering this distribution can help in directly forming the K'map from the truth table or the Σ notation.

YZ \ WX	00	01	11	10
00	m_0	m_4	m_{12}	m_8
01	m_1	m_5	m_{13}	m_9
11	m_3	m_7	m_{15}	m_{11}
10	m_2	m_6	m_{14}	m_{10}

In a 4 variable K'map, columns 1 and 4 are adjacent, so are rows 1 and 4

Different examples of 4 variable K'maps are shown in the next page.

The K'map shown on the right (**Ex. 48b**) takes into account the 'd' terms corresponding to the min-terms m_{10}, m_{11}, m_{12}, m_{13}, m_{14}, and m_{15} (the non-BCD terms) along with the normal output terms for example-48.

We find that by taking 5 of the 6 don't care terms as '1', we can **increase the size of the groups** thereby getting a more simplified output than **Ex. 48a** as shown in the last page. We have considered the d-term m_{12} as '0' and the remaining d-terms as '1'.

	$\bar{W}\bar{X}$	$\bar{W}X$	WX	$W\bar{X}$
$\bar{Y}\bar{Z}$			d	
$\bar{Y}Z$			d	1
YZ	1	1	d	d
$Y\bar{Z}$	1	1	d	d

Ex.48b: f = Y + WZ

Remember that you **should use the 'd' terms only to increase the size of an existing group**, if possible, and not to create new groups.

Worked out problems using K` maps

- Some worked out problems using K'maps:

Example-49*:* Using K' map find the simplified **POS** expression for the function: $f(X,Y,Z) = \Sigma\,(0,1,4,6)$

The first method is used to solve this. Therefore to find the POS expression, we have to first plot the complement of the output in the K' map.

The complement of the expression $f=\Sigma(0,1,4,6)$ will be $\bar{f}=\Sigma(2,3,5,7)$. $\bar{f}$ is plotted in the K'Map as shown on the right. The min term numbers are written in the lower-right corner of each cell in the K'map to help us plot the terms.

	$\bar{X}\bar{Y}$	$\bar{X}Y$	XY	$X\bar{Y}$
$\bar{Z}$	0	1 (2)	6	4
Z	1	1 (3)	1 (7)	1 (5)

$\bar{f}$	$\bar{X}\bar{Y}$	$\bar{X}Y$	XY	$X\bar{Y}$
$\bar{Z}$	0	1 (2)	6	4
Z	1	1 (3)	1 (7)	1 (5)

Group	Min terms taken	d-terms taken	Unchanged Term
1	m_2, m_3	none	$\bar{X}$, Y
2	m_5, m_7	none	X, Z

Therefore $\bar{f} = \bar{X}Y + XZ$

Or $f = \overline{(\bar{f})} = \overline{\bar{X}Y + XZ} = (\bar{\bar{X}} + \bar{Y}) \cdot (\bar{X} + \bar{Z}) = (X + \bar{Y}) \cdot (\bar{X} + \bar{Z})$

X	Y	Z	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	d
1	1	1	d

Example-50*:* A logic circuit implements the Boolean function: $\mathbf{f} = \bar{X}Y + X\bar{Y}\bar{Z}$. It is found that the combination X=Y=1 can never occur as input in the circuit. Find a simpler expression for **f**.

First we have to convert the expression to a canonical one to get all the valid min-terms. To do that we have to multiply the first term by $(Z+\bar{Z})$ to get:

$f = \bar{X}YZ + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z}$

	$\bar{X}\bar{Y}$	$\bar{X}Y$	XY	$X\bar{Y}$
$\bar{Z}$	0	1 (2)	d (6)	1 (4)
Z	1	1 (3)	d (7)	5

Group	Min terms taken	d-terms taken	Unchanged Term
1	m_2, m_3	m_6, m_7	Y
2	m_4	m_6	X, $\bar{Z}$

The truth table for the expression is shown on the right. From the truth table we find that the condition X=Y=1 occurs in the last two rows as shown by the highlighted box. As these input conditions will never occur, we can use these two combinations as don't-care terms. The output min-terms along with the '**d**' terms are plotted on the K' map as shown above. We can take both these d-terms as '1' and include them to increase the size of the required groups.

The final simplified output is given by $\mathbf{f} = Y + X\bar{Z}$

Example-51*:* It is necessary to design an overheat alarm for an oil-fired steam boiler. **Three sensors** are available. One sensor monitors the **water temperature** in the boiler, one monitors the **chimney temperature** and one follows the **on-off state of the burner**. The logic operations of the sensors are given below. An alarm signal should be generated whenever the **burner is on and either the chimney or the water temperature is too hot**. Obtain the Boolean expression for the alarm signal.

W	C	B	A	min
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	$\bar{W}CB$
1	0	0	0	
1	0	1	1	$W\bar{C}B$
1	1	0	0	
1	1	1	1	WCB

Boiler Water: '0' means water is within normal temperature range
'1' means water is too hot

Chimney: '0' means chimney is within normal temperature range
'1' means chimney is too hot

Oil Burner: '0' means burner is Off
'1' means burner is On

	$\bar{W}\bar{C}$	$\bar{W}C$	WC	$W\bar{C}$
$\bar{B}$	0	2	6	4
B	1	1	1	1

Solution:

Step 1: First we identify the different variables as:

w : the Water Temperature
C : the Chimney Temperature &
B : the Burner On-Off State

Step 2: Next construct the truth table for the problem and write down the desired outputs and their valid min terms as shown in the last page.

Step 3: Finally write down the Boolean expression from the truth table using the min-terms and simplify the same using K'map as shown on the right. Thus the final simplified expression is given as:

A = BC + BW

Example-52: A Boolean function is given in POS form as f (X,Y,Z) = Π (2, 3, 4, 5, 6). Find the simplified POS expression using a K'map.

We use the second method to find the simplified POS form of **f**. From the Π notation we directly plot the Max terms into the POS version of the K'map as shown on the right.

	$X+Y$	$X+\bar{Y}$	$\bar{X}+\bar{Y}$	$\bar{X}+Y$
Z	0	0	0	0
$\bar{Z}$	1	0	0	0

	$X+Y$	$X+\bar{Y}$	$\bar{X}+\bar{Y}$	$\bar{X}+Y$
Z	0	0	0	0
$\bar{Z}$	1	0	0	0

In this example we can **group the terms in two different ways** as shown in the two K'maps, thereby giving two different, but possible simplified outputs. Noting the terms which have not changed within each group and adding them, we finally have:

f = $(X+\bar{Y})(\bar{Y}+Z)(\bar{X}+Y)$ or **f** = $(X+\bar{Y})(\bar{X}+Z)(\bar{X}+Y)$

Example-53: Simplify the Boolean function **f** in SOP form using a K'map: **f**(W,X,Y,Z)=Σ(0,1,4,5,7,10,11,13,14,15)

The min term numbers are written in the lower-right corner of each cell in the K'map. The min-terms present in the Σ notation are directly plotted into a 4 variable K'map as shown on the right, by taking help of these min-term numbers.

	$\bar{W}\bar{X}$	$\bar{W}X$	WX	$W\bar{X}$
$\bar{Y}\bar{Z}$	1 0	1 4	12	8
$\bar{Y}Z$	1 1	1 5	1 13	9
YZ	3	1 7	1 15	1 11
$Y\bar{Z}$	2	6	1 14	1 10

	$\bar{W}\bar{X}$	$\bar{W}X$	WX	$W\bar{X}$
$\bar{Y}\bar{Z}$	1 0	1 4	12	8
$\bar{Y}Z$	1 1	1 5	1 13	9
YZ	3	1 7	1 15	1 11
$Y\bar{Z}$	2	6	1 14	1 10

We can form 3 groups as shown in the map, each containing 4 terms.

The required output is given by:

f = $\bar{W}\bar{Y} + XZ + WY$

Group	Min terms taken	d-terms taken	Unchanged Term
1	m_0, m_1, m_4, m_5	none	$\bar{W}, \bar{Y}$
2	m_5, m_7, m_{13}, m_{15}	none	X, Z
3	$m_{10}, m_{11}, m_{14}, m_{15}$	none	W, Y

Example-54: A Boolean function is given by: f(W,X,Y,Z) = Σ(0,1,5,13,14,15) with d = Σ(2,8,9,11). Find the simplified SOP expression using K'map.

The output terms are directly plotted into a 4-variable K'map using the min-term numbers. The don't care terms are also plotted.

As seen from the K'map, we can use the d-terms m_8 and m_9 to increase the size of a group. Hence only these are used in the K'map to form the groups.

	$\bar{W}\bar{X}$	$\bar{W}X$	WX	$W\bar{X}$
$\bar{Y}\bar{Z}$	1 0	4	12	d 8
$\bar{Y}Z$	1 1	1 5	1 13	d 9
YZ	3	7	1 15	d 11
$Y\bar{Z}$	d 2	6	1 14	10

	$\bar{W}\bar{X}$	$\bar{W}X$	WX	$W\bar{X}$
$\bar{Y}\bar{Z}$	1 0	4	12	d 8
$\bar{Y}Z$	1 1	1 5	1 13	d 9
YZ	3	7	1 15	d 11
$Y\bar{Z}$	d 2	6	1 14	10

The required output is given by:

f = $\bar{X}\bar{Y} + \bar{Y}Z + WXY$

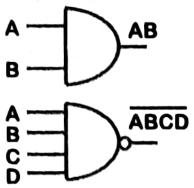
Group	Min terms taken	d-terms taken	Unchanged Term
1	m_0, m_1	m_8, m_9	$\bar{X}, \bar{Y}$
2	m_1, m_5, m_{13}	m_9	$\bar{Y}, Z$
3	m_{14}, m_{15}		WXY

K' map for POS expressions

4.9 Logic Gates: AND, OR, NOT, XOR, NAND, NOR, XNOR

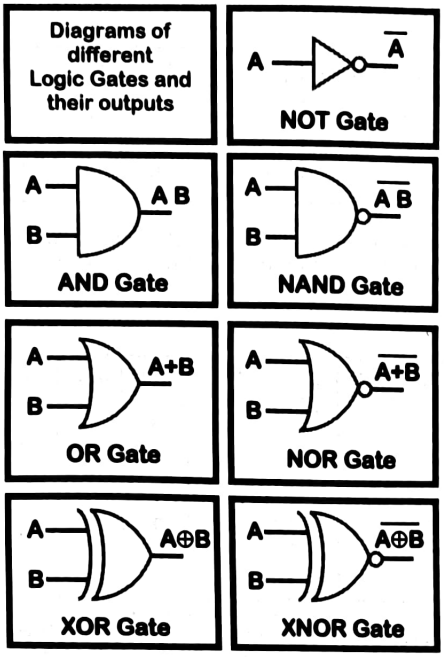
We have seen that digital computing is mainly based on binary logic, which operates with only 2 numbers i.e. '0' & '1' to represent the logic states 'False' and 'True' respectively. Modern circuits, which realise this logic, use **different voltage levels to represent the binary '0' and '1'**. Usually a 5V or a 12V DC supply is used to drive such ICs, with two different voltage ranges used to represent '0' and '1'.

When designed using electrical or electronic circuits, the **logic functions are represented by specific block diagrams known as logic gates**. The different functions are represented by different circuits and block diagrams. The diagram on the left shows the representation of a 2-input **AND logic gate**.



A logic gate has a single output whereas the inputs can vary from one (for NOT gate) to many. A 4-input NAND gate is shown on the left. The number of inputs is determined by the number of variables that are required to realise a function.

When designed using electrical or electronic circuits, the logic functions are represented by specific block diagrams known as Logic Gates



These logic gates form the basis of all combinational and sequential logic circuits. The logical components like CPU, Registers, RAM, etc. of computers are made up of different combinations of these logic gates. To design a circuit, first the truth table for the desired outputs for a set of given inputs is drawn. From the truth table the Boolean expression in SOP or POS form is then derived and simplified. Finally the logic gate representation of the circuit is made from the simplified expression.

The different logic gates corresponding to the different Boolean functions are shown on the left. The **complement operation** is represented by a triangle with a bubble at the end and is called a **NOT gate**. The other gates are shown with **2 inputs and 1 output**. The AND, OR and XOR gates have their corresponding complemented counterparts i.e. the NAND (**NOT AND**), the NOR (**NOT OR**) and the XNOR (**NOT XOR**). The **XOR** or **Exclusive OR** operation is represented by the **symbol** ⊕.

A XOR gate with inputs A and B, has the output given by: A⊕B, which is same as: $X\overline{Y} + \overline{X}Y$

The logic gate diagrams of the complement of all the basic gates are formed by **combining the un-complemented gate diagram with the bubble of the NOT gate** at the output side (see figure below).

The operations NAND, NOR, XOR and XNOR are new and hence their truth tables are given below:

A	B	$\overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

NAND

A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

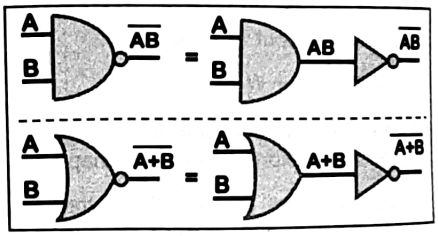
NOR

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

XOR

A	B	$\overline{A⊕B}$
0	0	1
0	1	0
1	0	0
1	1	1

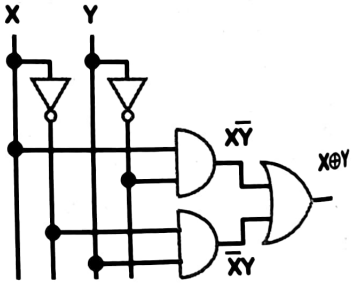
XNOR



Example-55: Draw the logic gate equivalent of the **XOR function** given by: $X⊕Y = \overline{X}Y + X\overline{Y}$

The given expression is an SOP one containing two product operations, one sum operation and two complement operations.

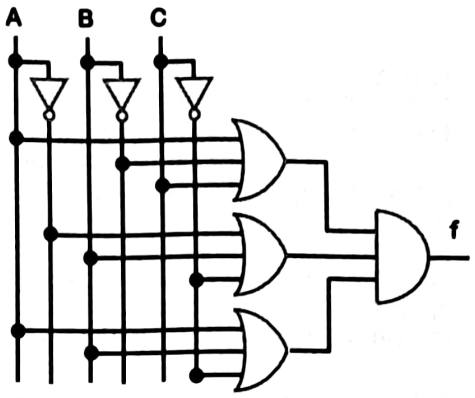
1. First get the **complement** of the variables **X** and **Y** using **NOT gates**, as the expression contains complemented variables
2. Next realise the product terms using two nos. of **2 input AND** gates for the two product terms $\overline{X}Y$ and $X\overline{Y}$
3. Finally use a **2 input OR gate** to get the final sum of the SOP expression i.e. $\overline{X}Y + X\overline{Y}$



First draw the different **inputs** as **parallel vertical lines**. Next draw the complements of the inputs at the side of the main input lines using **NOT** gates. In case a particular complement is not required then it is not drawn. **A dot is used to represent a connection between two lines**.

Example-56: Draw the logic gate equivalent of the Boolean function: $f = (A+\bar{B}+C)(\bar{A}+B+\bar{C})(A+B+\bar{C})$

The given expression is a POS one containing 3 sum terms, 2 product operations and 3 complement operations. To draw the required block diagram:



1. First step is to **get the complements of the variables using NOT gates**, as the expression contains complemented variables. (If the complements are available there is no need to carry out this step).
2. Next realise the **sum terms** using **three numbers of 3-input OR gates** for the three sum terms.
3. Finally use a **3 input AND gate** to get the **final product** of the POS expression.

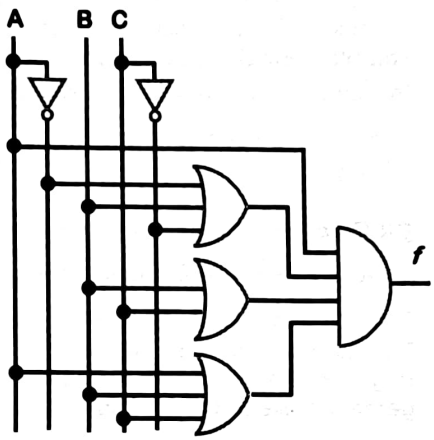
One thing to note in the above examples is that the complemented inputs to the AND & OR gates are taken from a single source, so that extra gates are not required to complement the inputs if a particular input is required more than once.

Example-57: Draw the logic gate equivalent of the Boolean function given by: $f = A\,(A+B+\bar{C})\,(B+C)\,(A+B+C)$

From the above expression it is seen that the complement of B i.e. $\bar{B}$ is not required and hence the same is not drawn (using a NOT gate) in this example.

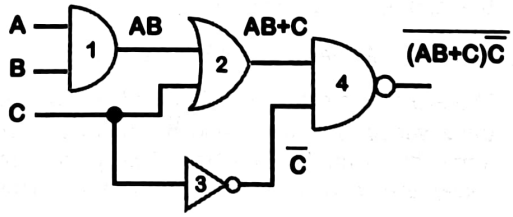
Moreover, the first term i.e. **A is directly connected** to the output AND gate. As it does not have any other term OR'ed to it, we have not used an OR gate for it. The same is applicable for an SOP expression also.

The SOP & POS circuits shown here are known as **two level logic networks**, as each input signal passes through **two** gates from input to output (assuming that complements of the inputs are available and hence not considering the NOT gates).



Example-58: The figure below shows a logic circuit composed of different gates. It has inputs A, B & C and final output *f*. It is required to find the final expression and hence simplify the circuit so that it can be realised using lesser number of gates.

The first step is to number the gates as shown in the figure. The output of each gate is then written just after the gate. The outputs of the gates are:



- **Gate 1**: Being an AND gate the output is AB
- **Gate 2**: Being an OR gate the output is (AB+C) since its two inputs are AB and C
- **Gate 3**: Being a NOT gate the output is $\bar{C}$
- **Gate 4**: Being a NAND gate, the output is given by $\overline{(AB + C)\,\bar{C}}$, since the inputs are (AB+C) and $\bar{C}$

On simplifying the output of the final NAND gate (gate 4) we get the final output as:

$f = \overline{(AB + C)\,\bar{C}} = \overline{AB\bar{C} + C\bar{C}} = \overline{AB\bar{C} + 0} = \overline{AB\bar{C}} = \bar{A}+\bar{B}+\bar{\bar{C}} = \bar{A}+\bar{B}+C$ [Using de-Morgan's theorem]

4.10 NAND and NOR Gates as Universal Logic Gates

Universal gates NAND, NOR

Of all the logic gates discussed so far, two gates occupy a special position. These are the **NAND** and **NOR** gates. These gates can be made in modern **IC** packages **using lesser number of components** (like

transistors) than the other gates. This **reduces the complexity, cost, and size of these devices.** All other gates can be made by proper combinations of either NAND or NOR gates only. Since **all types of logic gates can be formed by suitable combinations of either NAND or NOR logic gates only**, these are known as **Universal Gates**. We will now find the equivalent circuits for all other logic gates using universal gates only.

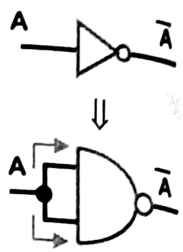
Other gates using NAND gates

NAND and NOR gates are called Universal Gates as any digital circuit can be formed using either NAND or NOR gates only

- **Realisation of Logic Components using NAND gates:**

NOT gate: From the truth table of the NAND function we see that **when the two inputs are same** (i.e. both are '0' or '1' as shown by the highlighted boxes in the truth table), the **output is the complement of the input**. To apply the same input value to both the inputs of the NAND gate, an easy solution is to **join both the inputs** of the NAND gate. Under such a condition if a single input is applied to the joined terminal, then that input will get distributed to both the input terminals (as shown by the arrows). Hence the **output will always be the complement of the input signal applied**. This is precisely what a NOT gate does. Hence the circuit is the **NAND gate equivalent of the NOT gate**.

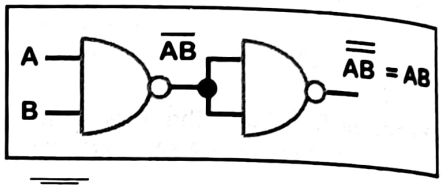
A	B	$\overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0



NAND Gate

AND using NAND

AND Gate: From the logic of an AND gate, if **A** and **B** are 2 variables then their AND operation will give the output $f = AB$. To make such a gate using NAND gates only, we have to write the expression of the AND gate to look like a NAND operation. To get that, the first thing to do is to **double complement the expression** as shown below. Then rearrange the terms so as to look like a NAND operation as: $f = A\,B = \overline{\overline{A\,B}} = \overline{(\overline{A\,B})}$



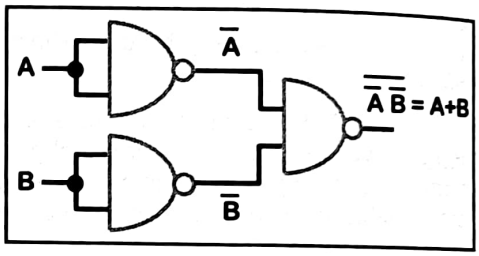
First use a NAND gate to get $\overline{AB}$ and then complement it using the equivalent NOT gate to get AB. Thus 2 NAND gates are required to form the AND as shown in the figure above.

OR using NAND

OR Gate: From the logic of an OR gate we have:

$f = A+B = \overline{\overline{A+B}} = \overline{\overline{A} \cdot \overline{B}}$ (using *de Morgan's Theorem*)

First complement the variables to get $\overline{A}$ & $\overline{B}$ using equivalent NOT gates and then use their outputs as inputs to a NAND gate to get the final output. Thus we require **3 numbers** of **NAND gates** to make an **OR function**.

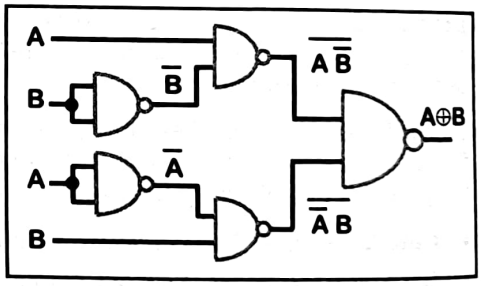


XOR using NAND

XOR Gate: To make an XOR gate first derive an expression that uses **only NAND operations** to get an XOR operation. How the NAND gates are to be connected is understood from that expression. We know that:

$f = A \oplus B = A\overline{B} + \overline{A}B = \overline{\overline{A\overline{B}} + \overline{\overline{A}B}} = \overline{(\overline{A\overline{B}})} \cdot \overline{(\overline{\overline{A}B})}$

The expression shows that first we have to complement each of the 2 variables to get $\overline{A}$ and $\overline{B}$. Then get each of the product terms by using NAND gates. Finally the product terms are joined using another NAND gate to get the final expression.

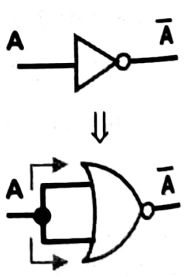


Other gates using NOR gates

- **Realisation of Logic Components using NOR gates:**

NOT gate: From the truth table of the NOR function we see that **when the two inputs are same** (i.e. both are '0' or '1' as shown by the highlighted boxes in the truth table), the **output is the complement of the input**. To apply the same input value to both the inputs of the NOR gate we **join both the inputs** of the NOR gate. Under such a condition if a single input is applied to the joined terminal, then that input will get distributed to both the input terminals. Hence the **output will always be the complement of the input signal applied**. Hence the circuit is the **NOR gate equivalent of the NOT gate**.

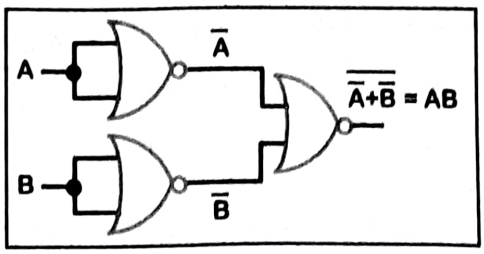
A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0



NOR Gate

NOT using NOR

AND gate: From the logic of an AND gate, for two variables **A** and **B**, their AND operation will give the output f = AB. To make such a gate using NOR gates only, we have to write the expression of the AND gate in terms of NOR operations. To arrive at such an expression first **double complement the expression** as shown below. Then rearrange the terms to look like a NOR operation.



AND using NOR

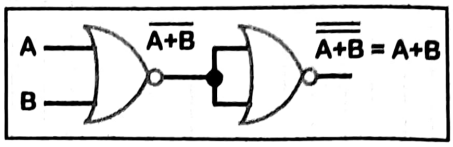
$f = AB = \overline{\overline{A\,B}} = \overline{\overline{A} + \overline{B}}$ (using *de Morgan's Theorem*)

First complement the variables to get $\bar{A}$ and $\bar{B}$ using equivalent NOT gates. Then input these to a NOR gate to get the final output. Thus we require **3 NOR gates** to get an **AND function**.

OR Gate: From the logic of an OR Gate we have:

$f = A+B = \overline{\overline{A+B}} = \overline{(\overline{A} + \overline{B})}$

We first need one NOR gate to get $(\overline{A+B})$ and then we complement it using the equivalent NOT gate to get A+B. Thus 2 NOR gates are required to form the OR as shown in the figure above.

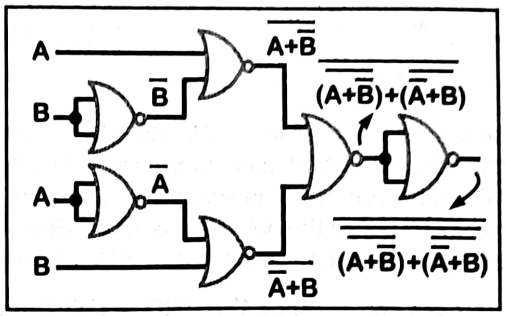


OR using NOR

XOR Gate: To make an XOR gate we first derive at an expression that **uses only NOR operations** to realise an XOR operation. From that expression we can know how the NOR gates are to be connected. We know that:

$f = A \oplus B = A\bar{B} + \bar{A}B = \overline{\overline{A\bar{B}} + \overline{\bar{A}B}} = \overline{(\overline{A\bar{B}})\cdot(\overline{\bar{A}B})} = \overline{(\bar{A}+B)\cdot(A+\bar{B})}$

$= \overline{\bar{A}+B} + \overline{A+\bar{B}} = \overline{\overline{\bar{A}+B}} + \overline{\overline{A+\bar{B}}} = \left[\overline{(\bar{A}+B)} + \overline{(A+\bar{B})}\right]$



XOR using NOR

The expression shows that first we have to complement each of the 2 variables to get $\bar{A}$ and $\bar{B}$. Then get each of the sum terms $\overline{\bar{A}+B}$ and $\overline{A+\bar{B}}$ by using 2 NOR gates. Next create the term $\overline{\overline{\bar{A}+B} + \overline{A+\bar{B}}}$ by using another NOR gate. Finally the output is achieved by complementing this expression using a NOT gate.

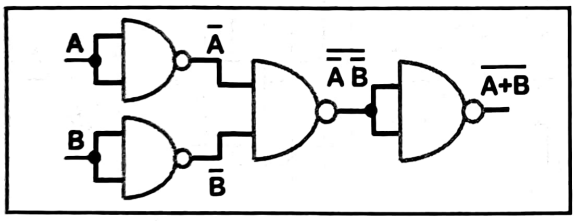
- **NOR gate using NAND gate and NAND gate using NOR gate:**

To get the NAND equivalent of NOR, we express the NOR operation in a NAND format as:

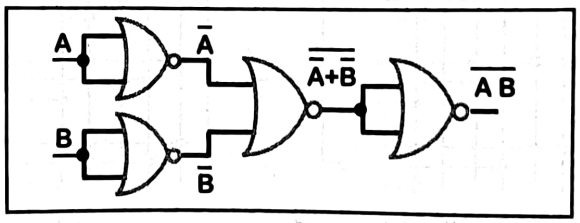
NOR = $\overline{A+B} = \bar{A}\,\bar{B} = \overline{\bar{A}\,\bar{B}} = \overline{(\bar{A}\cdot\bar{B})}$

To get the NOR equivalent of NAND, we express the NAND operation in a NOR format as:

NAND = $\overline{A\cdot B} = \bar{A} + \bar{B} = \overline{\bar{A} + \bar{B}} = \overline{(\bar{A} + \bar{B})}$



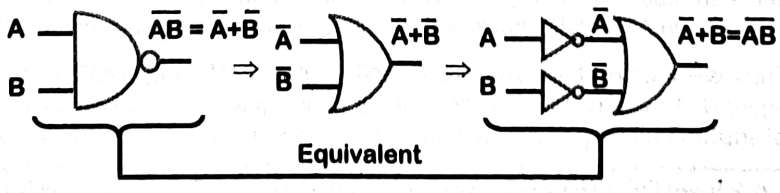
NOR gate using NAND gates



NAND gate using NOR gates

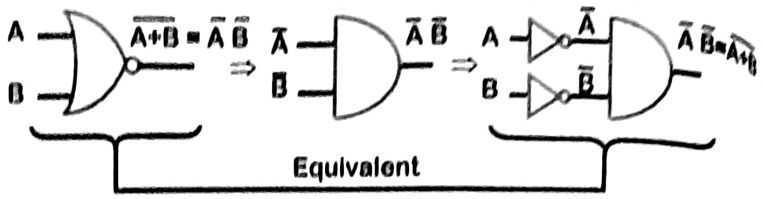
- **Design Rules Using NAND and NOR Gates:**

Using De Morgan's theorem a **NAND** function can be written as: $\overline{A\cdot B} = \bar{A} + \bar{B}$ i.e. as an **OR** operation with the inputs A and B complemented. See the diagram on the right which shows the equivalent circuit.



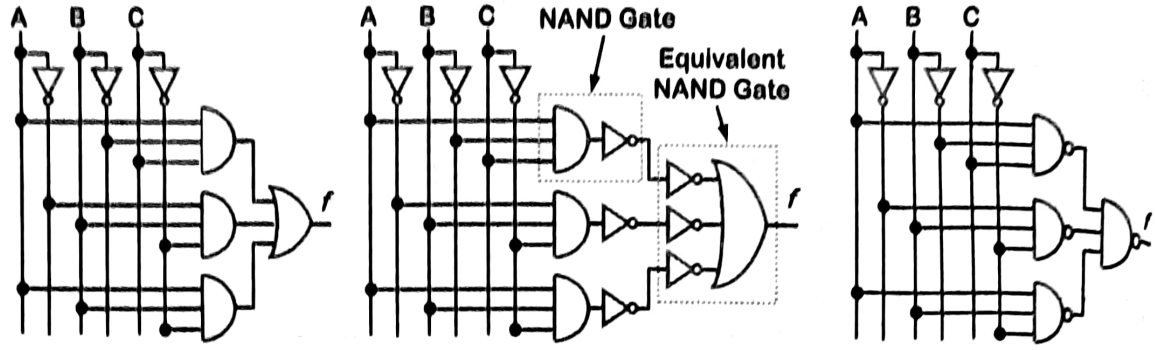
Design Rules using NAND and NOR gates

Similarly a **NOR** function can be written as: $\overline{A+B} = \overline{A} \cdot \overline{B}$ i.e. as an **AND** operation with the inputs A and B complemented. See the diagram on the right which shows the equivalent circuit.



This leads to the **equivalent circuits** for the NAND and NOR gates as shown in the diagrams above. The NOT gates at the input of the OR and AND gates are used to complement the inputs A and B before they enter the OR and AND gates.

Two level NAND-NAND circuit



a) AND-OR Circuit ⟵ Same as... ⟶ b) NAND-NAND Circuit

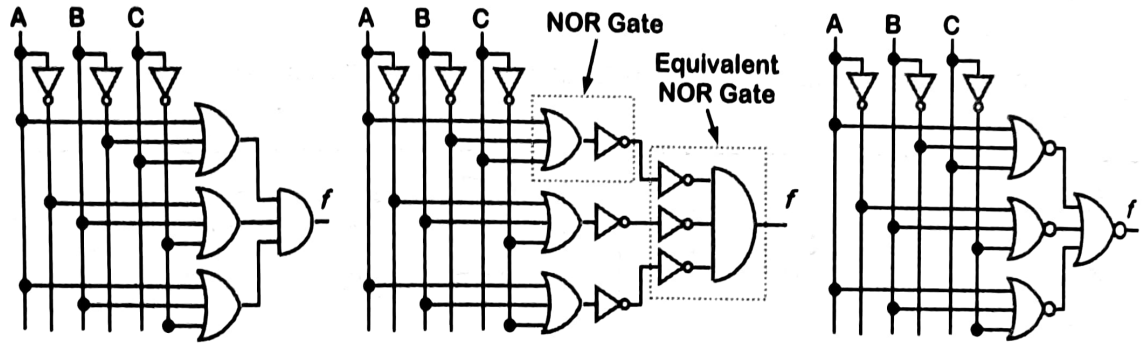
A two level AND-OR is same as a two level NAND-NAND

Now consider the 2-level AND-OR **SOP circuit** shown in fig. **a** above, consisting of 3 AND gates and an OR gate. We add a NOT gate after each AND gate and a NOT gate before each input line to the OR gate. Since 2 NOT gates are added in series, the circuit remains unaffected (since double NOT gate means complementing twice and hence the net effect is zero). However an AND followed by a NOT is simply a NAND gate and an OR with gate complemented inputs is same as an equivalent NAND gate as has been shown just now.

Thus **we can replace the AND and OR gates with only NAND gates** and derive at the circuit shown in figure **b**. Thus a two level NAND-NAND circuit is equivalent to a two level AND-OR circuit with an **identical set of inputs** and output. Accordingly an SOP expression can be directly formed using NAND gates only.

A 2-level AND-OR SOP circuit is equivalent to a 2-level NAND-NAND circuit with identical inputs and hence an SOP circuit can be directly formed using NAND gates only

Two level NOR-NOR circuit



c) OR-AND Circuit ⟵ Same as... ⟶ d) NOR-NOR Circuit

A two level OR-AND is same as a two level NOR-NOR

Next consider the 2-level OR-AND **POS circuit** shown in fig. **c** consisting of 3 OR gates and an AND gate. We add a NOT gate after each OR gate and a NOT gate before each input line to the AND gate. Since the 2 NOT gates are added in series, the circuit remains unaffected. However an OR followed by a NOT is simply a NOR gate and an AND with complemented inputs is same as an equivalent NOR gate as has been shown before.

Thus **we can replace the OR & AND gates with only NOR gates** and derive at the circuit shown in figure **d**. Thus a two level NOR-NOR circuit is equivalent to a two level OR-AND circuit with an **identical set of inputs** and output. Accordingly a POS expression can be directly formed using NOR gates only.

A 2-level OR-AND POS circuit is equivalent to a 2-level NOR-NOR circuit with identical inputs and hence a POS circuit can be directly formed using NOR gates only

Example-59: A 3-input logic circuit with inputs A, B, C generates a high output under following conditions:

- A=0, B=0
- A=0, B=1, C=1
- A=1, B=0, C=1
- A=1, B=1, C=1

A	B	C	f	min
0	0	0	1	$\bar{A}\bar{B}\bar{C}$
0	0	1	1	$\bar{A}\bar{B}C$
0	1	0	0	
0	1	1	1	$\bar{A}BC$
1	0	0	0	
1	0	1	1	$A\bar{B}C$
1	1	0	0	
1	1	1	1	ABC

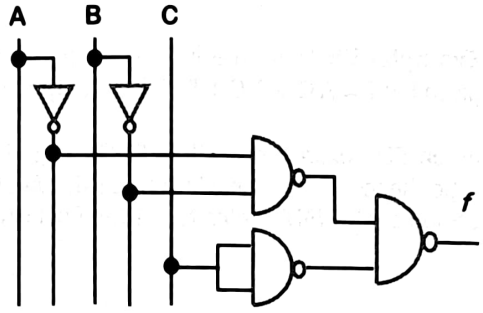
Draw the truth table, find the SOP expression and simplify the same and design a combinational circuit for this system using NAND gates only.

The truth table is shown on the right. Note that for the first condition i.e. A=0 and B=0, since nothing is mentioned about C, we have taken both the values of C along with A=0 and B=0, i.e. we take both the inputs A=0, B=0, C=0 and A=0, B=0, C=1 to satisfy the first condition. Taking the valid min terms from the truth table, the required SOP expression is:

$$f = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}BC + A\bar{B}C + ABC$$

The required simplified SOP expression is:

$$\begin{aligned} f &= \bar{A}\bar{B}(\bar{C} + C) + \bar{A}BC + ABC + A\bar{B}C + ABC \\ &= \bar{A}\bar{B} + (\bar{A} + A)BC + AC(\bar{B} + B) = \bar{A}\bar{B} + BC + AC \\ &= (\overline{A+B}) + (A+B)C = (\overline{A+B}) + C = \bar{A}\bar{B} + C \end{aligned}$$



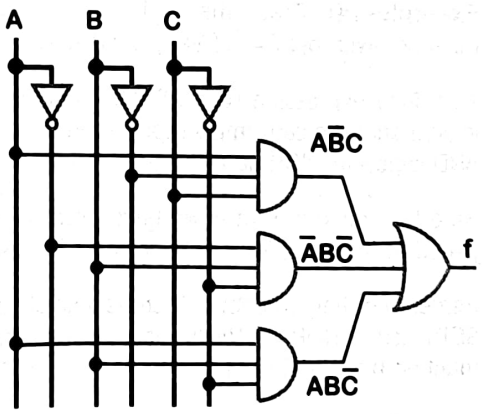
Using the design rules for AND-OR circuit (same as NAND-NAND) we draw the required circuit diagram using only NAND gates. Note that for the single input C, we have used a NAND gate with inputs joined.

4.11 Worked out problems on Logic Gates

Worked out problems using logic gates

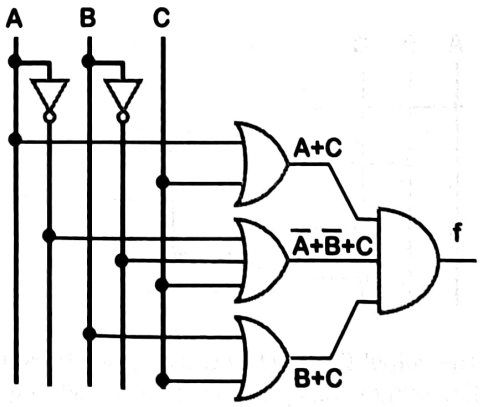
Example-60: Draw the logic gate equivalent of the Boolean function given by: $f = A\bar{B}C + \bar{A}B\bar{C} + AB\bar{C}$

Each AND gate forms an output corresponding to a min term by logically multiplying the inputs A, B, and C. The outputs of all the AND gates are taken as inputs for the OR gate, which logically adds the min terms to form the required SOP output.



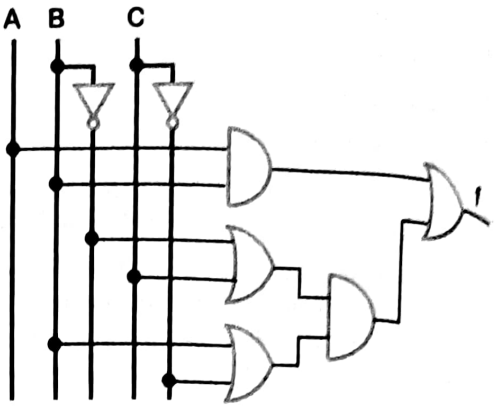
Example-61: Draw the logic gate equivalent of the Boolean function given by: $f = (A+C)(\bar{A}+\bar{B}+C)(B+C)$

Each OR gate forms an output corresponding to a max term by logically adding the inputs A, B, and C. The outputs of all the OR gates are taken as inputs for the AND gate, which logically multiplies the max terms to form the required POS output.



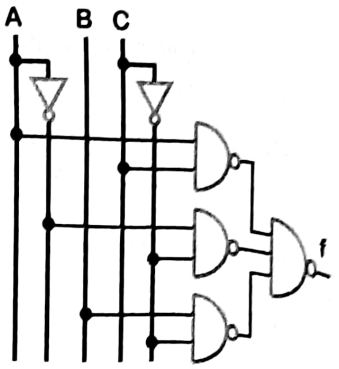
Example-62: Draw the logic gate equivalent of the Boolean function given by: $f = AB + (\bar{B}+C)(B+\bar{C})$

The first AND gate is used to get the product AB. The two OR gates are used to get the terms $(\bar{B}+C)$ and $(B+\bar{C})$. The second AND gate is used to multiply the terms $(\bar{B}+C)(B+\bar{C})$. The final OR gate is used to get the logical sum of the terms AB and $(\bar{B}+C)(B+\bar{C})$ to get the final sum.



Example-63: Draw the logic gate equivalent of the Boolean function given by: $f = A\,C + \bar{A}\,\bar{C} + B\,\bar{C}$ using only NAND gates.

As an SOP expression with AND-OR logic is equivalent to NAND-NAND logic, hence we can simply replace the AND and OR gates of the AND-OR logic with NAND gates to get the final circuit as shown on the right.

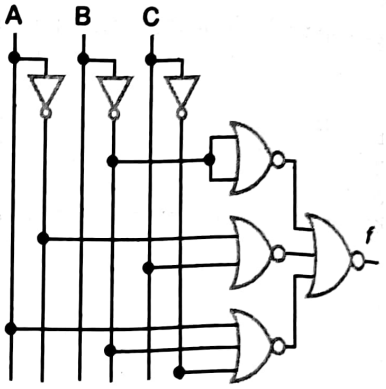


Example-64: Draw the logic gate equivalent of the Boolean function given by: $f = \bar{B}\,(\bar{A}+C)\,(A+\bar{B}+\bar{C})$ using only NOR gates.

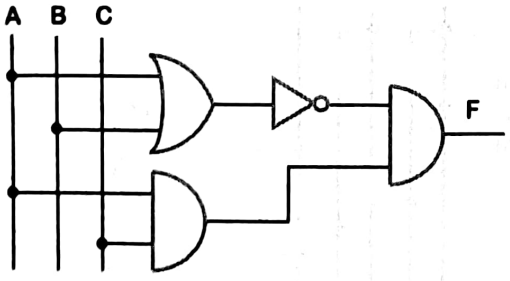
As a POS expression with OR-AND logic is equivalent to NOR-NOR logic, hence we can simply replace the OR and AND gates of the OR-AND logic with NOR gates to get the final circuit.

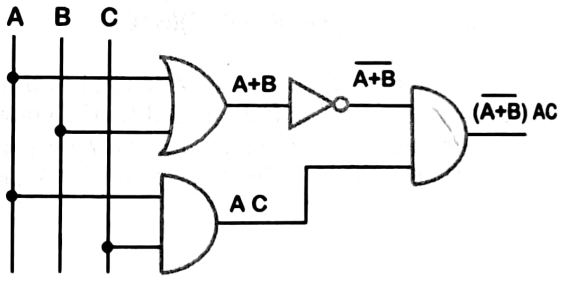
Note that for the term $\bar{B}$ we have used a NOR gate with the inputs joined to get the 2-level NOR-NOR equivalent circuit.

Remember that, whenever there is a single input (either in POS or in SOP) use a NOR or NAND gate as per the type of circuit in this manner in the first level.



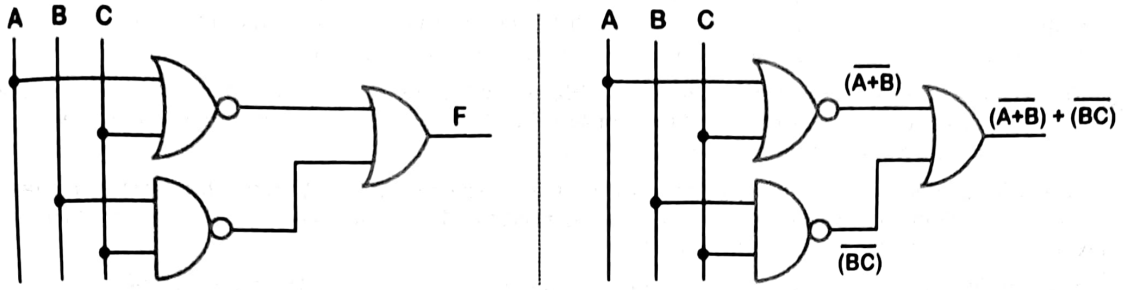
Example-65: Find the output F of the following logic circuit:





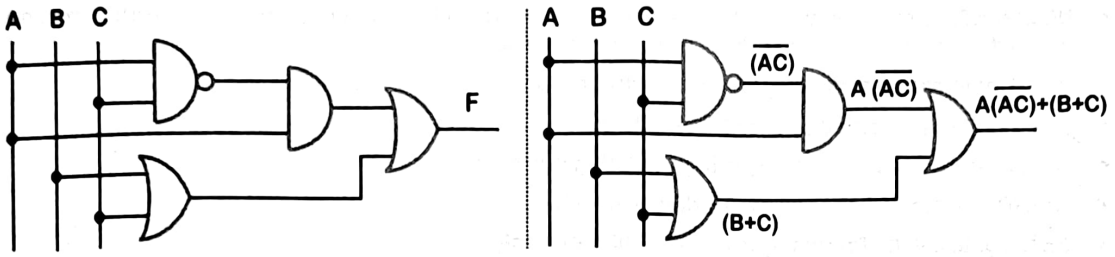
The output **F** = $\overline{(A+B)}$ AC is obtained by writing the output of each of the logic gates individually starting from the leftmost gate, as shown in the second diagram, keeping in mind the output produced by each type of logic gate.

Example-66: Find the output F of the following logic circuit:



The output $F = (\overline{A+B}) + \overline{B}\,\overline{C}$ is obtained by writing the output of each of the logic gates individually starting from the leftmost gate, as in the previous example.

Example-67: Find the output F of the following logic circuit, and simplify the output.



$F = A(\overline{AC}) + (B+C) = A(\overline{A}+\overline{C}) + (B+C) = A\overline{A} + A\overline{C} + B+C = A\,\overline{C}+B+C = C + \overline{C}\,A + B = C+A+B$

$= A+B+C$

Fact File

- **Boolean algebra was developed by the English mathematician George Boole. His work was based on the analysis of logic and is contained in his book named "An investigation of the laws of thought on which are founded the Mathematical Theories of Logic and Probabilities".**
- OR operation is designated by the '+' operator and indicates a Logical Addition. The logical output is written as: C = A+B and is read as 'C equals A OR B' or 'C equals A plus B'.
- **AND operation is designated by the '•' (dot) operator and indicates a Logical Multiplication. The logical output is written as: C = A•B and is read as 'C equals A AND B' or 'C equals A B'.**
- The NOT or complement operation is designated by a '¯' (bar) placed over the variable and is used to indicate a Logical Complement. It is a unary operation i.e. needs a minimum of 1 variable to act upon. The logical output is written as: $C = \overline{A}$ and is read as "C is the complement of A".
- **Two switches A & B connected in parallel function like OR logic with the output high whenever any one of the switches or both the switches are closed.**
- **Two switches A & B connected in series function like AND logic where the output will be high only when both the switches are closed.**
- De Morgan's Sum Theorem: The complement of the logical sum of two variables X and Y is equal to the logical product of the individual complements of the variables. Thus $\overline{X+Y} = \overline{X} \cdot \overline{Y}$
- **De Morgan's Product Theorem: The complement of the logical product of two variables X and Y is equal to the logical sum of the individual complements of the variables. Thus $\overline{X \cdot Y} = \overline{X} + \overline{Y}$**
- The general rule to form an SOP expression is as follows: Wherever the desired output is 1, take the min-terms for those outputs and add those min-terms to get the final result
- **The general rule to form a POS expression is as follows: Wherever the desired output is 0, take the max-terms for those outputs and multiply those max-terms to get the final result.**
- $\overline{\Sigma(m_j)} = \Pi(M_j)$ and $\overline{\Pi(M_j)} = \Sigma(m_j)$
- When a Boolean expression in the SOP form has product terms each of which contains all the variables (either in complemented or in non-complemented form) then the expression is called a Canonical SOP
- ***If a Boolean expression in the POS form contains sum terms each containing all the variables (either in complemented or in non-complemented form) then the expression is called a Canonical POS***

- To convert a non-canonical SOP to canonical form, first multiply the non-canonical term with an expression of the form $(X+\overline{X})$. Choose the multiplier in such a manner that it contains **that** variable which is **not** present in the original term. Repeat the process until each term is converted to its canonical form
- **To convert a non-canonical POS to canonical POS, add to the non-canonical term an expression of the form $X\,\overline{X}$. Apply the distributive rule $X+YZ = (X+Y)(X+Z)$. Repeat the process until each term is converted to its canonical form**
- To simplify an SOP expression using a K'map, include the maximum number of 1's within the minimum number of groups until all the '1's are covered, with a group containing 8, 4, 2, or a single term. Diagonal grouping is not allowed and a group can be only rectangular in shape.
- **A K'map for SOP expressions repeatedly use the Boolean rule $X + \overline{X} = 1$, to do the simplification**
- A K'map for POS expressions repeatedly use the Boolean rule $X \cdot \overline{X} = 0$, to do the simplification
- **The logic functions when realised using electrical or electronic circuits, they are represented on paper by specific block diagrams known as logic gates. The different functions are represented by different block diagrams**
- NAND & NOR gates are known as Universal Gates since all types of gates can be formed by combining these gates only and hence any circuit can be designed using these gates only
- **A NOT gate can be formed by using 1 NAND gate only**
- A NOT gate can be formed by using 1 NOR gate only
- **An AND gate can be formed by using 2 NAND gates only**
- An AND gate can be formed by using 3 NOR gates only
- **An OR gate can be formed by using 3 NAND gates only**
- An OR gate can be formed by using 2 NOR gates only
- **A NOR gate can be formed by using 4 NAND gates only**
- A NAND gate can be formed by using 4 NOR gates only
- **The NAND & NOR gates do not obey the associative rule and we cannot get the complement of the product or sum of 3 variables using two 2-input NAND or NOR gates only. To get a 3-input NAND gate from 2-input NAND gates we have to use three NAND gates**
- A 2-level AND-OR SOP circuit is equivalent to a 2-level NAND-NAND circuit with identical inputs and hence an SOP circuit can be directly formed using NAND gates only
- **A 2-level OR-AND POS circuit is equivalent to a 2-level NOR-NOR circuit with identical inputs and hence a POS circuit can be directly formed using NOR gates only**

Review Questions

Q1. **Fill in the blanks:** **1 each**

a) George Boole is the founder of Boolean Algebra.

b) OR and AND are the basic operations of Boolean Algebra.

c) The simplified value of X+XY is X.

d) The complement of $(X+\overline{Y})$ is $\overline{X}+Y$.

e) In Boolean algebra $(X+\overline{X}Y)$ = X+Y.

f) By using the distributive law, $(X+\overline{Y})(X+Y)$ will be equal to X.

g) The simplified value of X(X+Y) will be XY.

h) A truth table is a tabular representation of a logical problem.

i) With 'n' number of binary variables we can have _____ different combinations.

j) The full form of SOP is _____.

k) A term in a canonical SOP is called a _____ term.

l) The canonical expression for $f(X,Y,Z) = \overline{X}Y$ will be _____.

m) The full form of POS is _____.

n) A term in a canonical POS is called a _____ term.

o) $(A+\overline{C})\,(A+\overline{B}+\overline{C})\,(B+\overline{C})$ is an example of a _____ POS expression.

p) F(A,B,C) = Σ (0, 4, 7) is same as Π (_____).

q) F(X,Y,Z) = Π (2, 3, 5) is same as Σ ( ).

r) The simplified value of f(A,B,C) = Σ(2,3,6,7) is _____.

Q2. **Multiple Choice Questions. Select any one from the four options.** **1 each**



i) In Boolean algebra for the variable A, the value of A+A+A is equal to:
a. 1 b. 3A c. A d. A^3

ii) In Boolean algebra for the variable A, the value of A.A.A is equal to:
a. A b. 0 c. A^3 d. 3A

iii) In Boolean algebra for the variable A, the value of A+1 is equal to:
a. A b. 1 c. 0 d. $\bar{A}$

iv) In Boolean algebra for the variable A, the value of A+$\bar{A}$ is equal to:
a. $\bar{A}$ b. 0 c. 1 d. A

v) In Boolean algebra for the variable A, the value of A.$\bar{A}$ is equal to:
a. 1 b. A c. $\bar{A}$ d. 0

vi) In Boolean algebra for the variable A, the value of $\bar{\bar{A}}$ is equal to:
a. 0 b. 1 c. A d. $\bar{A}$

vii) In Boolean algebra X+$\bar{X}$Y is equal to:
a. X+$\bar{Y}$ b. $\bar{X}$+Y c. XY d. X+Y

viii) In Boolean algebra the distributive law (X+Y)(X+Z) is equal to:
a. X+YZ b. X+Y+Z c. XYZ d. XY+XZ

ix) In Boolean algebra the relation A.A=A is called:
a. Associative Law b. Commutative Law c. Idempotent Law d. Distributive Law

x) In Boolean algebra the relation A+B=B+A is called:
a. Associative Law b. Distributive Law c. Idempotent Law d. Commutative Law

xi) In Boolean algebra the relation X+XY=X is called:
a. Absorptive Law b. Idempotent Law c. Distributive Law d. Associative Law

xii) In Boolean algebra the relation X+(Y+Z) = (X+Y)+Z is called:
a. Distributive Law b. Associative Law c. Absorptive Law d. Idempotent Law

xiii) Using the distributive law the value of (X+$\bar{Y}$)(X+Y) is equal to:
a. XY b. $\bar{Y}$ c. X d. X+Y

xiv) Parallel electric switches in a circuit are an example of:
a. logical AND b. logical NOT c. logical OR d. None of these

xv) Series electric switches in a circuit are an example of:
a. logical OR b. logical AND c. logical NOT d. None of these

xvi) Water taps connected in parallel in different pipes are an example of:
a. logical AND b. logical OR c. logical NOT d. None of these

xvii) Water taps connected one after the other in the same pipe are an example of:
a. logical OR b. logical NOT c. logical AND d. None of these

xviii) In Boolean algebra X Y + $\bar{X}$ Y is equal to:
a. X+Y b. X c. 1 d. Y

xix) In Boolean algebra A+AB is equal to:
a. AB b. A c. B d. 1

xx) In Boolean algebra A + A $\bar{B}$ + $\bar{B}$ is equal to:
a. A + $\bar{B}$ b. A + B c. $\bar{A}$ + B d. $\bar{A}$ + $\bar{B}$

xxi) In Boolean algebra A B + $\bar{B}$ is equal to:
a. $\bar{A}$ + B b. A + B c. A + $\bar{B}$ d. $\bar{A}$ + $\bar{B}$

xxii) In Boolean algebra X+XY+Y is equal to:
a. $\bar{X}+Y$ b. $\bar{X}+\bar{Y}$ c. $X+\bar{Y}$ d. $X+Y$

xxiii) In Boolean algebra $XY+YZ+X+Z$ is equal to:
a. $X+Z$ b. $\bar{X}+\bar{Z}$ c. $X+\bar{Z}$ d. $\bar{X}+Z$

xxiv) In Boolean algebra X+XY+YZ is equal to:
a. X+YZ b. Y+ZX c. Z+XY d. X+Y+Z

xxv) In Boolean algebra $AC+A\bar{B}C$ is equal to:
a. AB b. BC c. AC d. A+BC

xxvi) In Boolean algebra $(X+Y\bar{Z})(X+\bar{Y}Z)$ is equal to:
a. Z b. $\bar{Z}$ c. X d. $\bar{X}$

xxvii) In Boolean algebra $\overline{XZ+\bar{Z}X}$ is equal to:
a. X b. $\bar{X}$ c. $\bar{Z}$ d. Z

xxviii) In Boolean algebra $\overline{(X+Z)(\bar{Z}+X)}$ is equal to:
a. Z b. $\bar{Z}$ c. X d. $\bar{X}$

xxix) In Boolean algebra $\overline{XZ+Z+1}$ is equal to:
a. 0 b. X c. 1 d. Z

xxx) In Boolean algebra $\overline{Z+\bar{Z}X}$ is equal to:
a. $\bar{X}Z$ b. $\bar{X}\bar{Z}$ c. $X\bar{Z}$ d. XZ

xxxi) In Boolean algebra, the complement of $(X+\bar{Y})$ is:
a. $\bar{X}\bar{Y}$ b. $\bar{X}Y$ c. XY d. $X\bar{Y}$

xxxii) In Boolean algebra $F(A,B,C) = \Sigma(0, 4, 7)$ is same as:
a. $\Pi(1,2,3,5,6)$ b. $\Pi(0,4,7)$ c. $\Pi(0,2,4,6)$ d. $\Pi(1,3,5,7)$

xxxiii) In Boolean algebra $F(A,B,C) = \Sigma(1, 2, 5, 6)$ is same as:
a. $\Pi(1,2,5,6)$ b. $\Pi(0,4,6,7)$ c. $\Pi(0,1,4,6)$ d. $\Pi(0,3,4,7)$

xxxiv) In Boolean algebra $F(A,B,C) = \Pi(1, 3, 5, 7)$ is same as:
a. $\Sigma(0, 3, 5, 6)$ b. $\Sigma(0, 1, 6)$ c. $\Sigma(0, 2, 4, 6)$ d. $\Sigma(2, 5, 6)$

xxxv) In Boolean algebra $F(A,B,C) = \Pi(2,6,7)$ is same as:
a. $\Sigma(0, 4, 5)$ b. $\Sigma(0, 1, 3, 4, 5)$ c. $\Sigma(0, 1, 2, 5, 6)$ d. $\Sigma(1, 2, 3, 5, 7)$

xxxvi) To convert a non-canonical SOP term to canonical form it is to be multiplied by a term like:
a. $X+\bar{X}$ b. $X.\bar{X}$ c. $X+\bar{Y}$ d. $\bar{X}+1$

xxxvii) To convert a non-canonical POS term to canonical form it is to be added to a term like:
a. $X+\bar{X}$ b. $\bar{X}+1$ c. $X+\bar{Y}$ d. $X.\bar{X}$

xxxviii) In Boolean algebra the canonical form of the expression f(X,Y)=XY+X+Y is equal to:
a. $XY+X\bar{Y}+\bar{X}Y$ b. $X\bar{Y}+\bar{X}Y$ c. $XY+\bar{X}Y$ d. $\overline{X\bar{Y}}+X\bar{Y}+\bar{X}Y$

xxxix) In Boolean algebra the canonical form of the expression f(X,Y,Z)=XY+XYZ is equal to:
a. $\bar{X}YZ+XY\bar{Z}$ b. $XYZ+XY\bar{Z}$ c. XYZ+XY d. $XYZ+X\overline{Y\bar{Z}}+XY\bar{Z}$

xl) In Boolean algebra the canonical form of the expression f(X,Y)=X'Y+Y' is equal to:
a. X' Y+X Y'+X' Y' b. X Y+X Y'+X' Y' c. X' Y+X Y d. X' Y+X Y'+X Y

xli) In Boolean algebra the canonical form of the expression $f(X,Y)= (\bar{X}+Y)X$ is equal to:
a. $(\bar{X}+Y)(\bar{X}+\bar{Y})(X+\bar{Y})$ b. $(\bar{X}+Y)(X+\bar{Y})$ c. $(X+Y)(X+\bar{Y})$ d. $(\bar{X}+Y)(X+Y)(X+\bar{Y})$

xlii) In Boolean algebra the canonical form of the expression f(X,Y,Z)=(X+Y')(X+Z) is equal to:
a. (X+Y+Z)(X'+Y'+Z)(X+Y'+Z') b. (X+Y+Z)(X+Y'+Z)(X'+Y'+Z')
c. (X+Y+Z)(X+Y'+Z)(X+Y'+Z') d. (X'+Y+Z)(X+Y'+Z)(X+Y'+Z')

xliii) In Boolean algebra the canonical form of the expression f(X,Y,Z)=(X+Y)(X+Y+Z) is equal to:
a. $(X+\bar{Y}+Z)(X+Y+\bar{Z})$ b. (X+Y+Z)(X+Y+Z) c. (X+Y+Z) d. $(\bar{X}+Y+Z)(X+\bar{Y}+Z)$

xliv) Which of the following is a valid statement for De Morgan's theorems:
a. $\overline{A\bar{B}}=A+B$ b. $\overline{A\bar{B}}=A.B$ c. $\overline{A\bar{B}}=\bar{A}+\bar{B}$ d. $\overline{A\bar{B}}=\bar{A}.\bar{B}$

xlv) Which of the following is a valid statement for De Morgan's theorems:
a. $\overline{A+B}=A+B$ b. $\overline{A+B}=A.B$ c. $\overline{A+B}=\overline{A}+\overline{B}$ d. $\overline{A+B}=\overline{A}.\overline{B}$

xlvi) The full form of SOP is:
a. Sum or Product b. Sum on Product c. Sum off Product d. Sum of Product

xlvii) The full form of POS is:
a. Product of Sum b. Product or Sum c. Product on Sum d. Product off Sum

xlviii) The value of a min term in Boolean algebra is:
a. 0 b. 1 c. 2 d. 3

xlix) The value of a max term in Boolean algebra is:
a. 3 b. 2 c. 1 d. 0

l) 4 Boolean variables can form how many different combinations?
a. 4 b. 8 c. 16 d. 32

li) The simplified form of the Boolean expression $f(X,Y,Z)=\Sigma(2,3,6,7)$ is:
a. $\overline{Y}$ b. X c. Y d. $\overline{X}$

lii) The simplified form of the Boolean expression $f(X,Y,Z)=\Sigma(0,1,4,5)$ is:
a. $\overline{Y}$ b. X c. Y d. $\overline{X}$

liii) The simplified form of the Boolean expression $f(X,Y,Z)=\Sigma(0,1,2,3)$ is:
a. $\overline{Y}$ b. X c. Y d. $\overline{X}$

liv) The simplified form of the Boolean expression $f(X,Y,Z)=\Sigma(4,5,6,7)$ is:
a. $\overline{Y}$ b. Y c. X d. $\overline{X}$

lv) The simplified form of the Boolean expression $f(X,Y,Z)=\Pi(0,1,2,3)$ is:
a. X b. $\overline{Y}$ c. $\overline{X}$ d. Y

lvi) The simplified form of the Boolean expression $f(X,Y,Z)=\Pi(0,2,4,6)$ is:
a. $\overline{Z}$ b. $\overline{Y}$ c. Z d. Y

lvii) The simplified form of the Boolean expression $f(X,Y,Z)=\Pi(2,3,6,7)$ is:
a. Z b. Y c. $\overline{Z}$ d. $\overline{Y}$

lviii) The simplified form of the Boolean expression $f(X,Y,Z)=\Pi(0,1,4,5)$ is:
a. $\overline{Y}$ b. Z c. $\overline{Z}$ d. Y

lix) Logic functions are represented by specific block diagrams known as:
a. logic circuits b. logic diagrams c. logic gates d. logic blocks

lx) A logic gate has how many outputs?
a. 0 b. 2 c. 3 d. 1

lxi) Which logic gate has a single output?
a. XNOR b. NOR c. NAND d. NOT

lxii) The logic gate represented by a triangle with a bubble at the tip is called a:
a. NOR gate b. XNOR gate c. NOT gate d. NAND gate

lxiii) The logic gate diagrams of the complement of all the basic gates are formed by combining the uncomplemented gate diagram with a small _ at the output side:
a. triangle b. circle c. square d. ellipse

lxiv) The complement of the AND gate is called a:
a. NAND b. NOT c. NOR d. XNOR

lxv) The complement of the OR gate is called a:
a. NOT b. NOR c. XNOR d. NAND

lxvi) The Exclusive OR operation is represented by the symbol:
a. + b. $\oplus$ c. θ d. $\otimes$

lxvii) Which of the following gates is called a universal gate?
a. XNOR b. NOT c. NAND d. XOR

bxviii) All types of logic gates can be formed by suitable combinations of _____ gates only:
a. XOR b. NOR c. XNOR d. OR

bxix) NAND gate is called a:
a. Uniform Gate b. Unilateral Gate c. Universal Gate d. Unidigital Gate

bxx) How many NAND gates are required to form a NOT gate?
a. 4 b. 3 c. 2 d. 1

bxxi) How many NAND gates are required to form an AND gate?
a. 4 b. 3 c. 2 d. 1

bxxii) How many NAND gates are required to form an OR gate?
a. 4 b. 3 c. 2 d. 1

bxxiii) The output of a 2 input XOR gate with inputs A and B is given by:
a. AB b. $\bar{A}\,\bar{B} + A\,B$ c. $A\bar{B} + \bar{A}B$ d. A+B

bxxiv) The output of a 2 input XNOR gate with inputs A and B is given by:
a. $\bar{A}\,\bar{B} + A\,B$ b. AB c. A+B d. $A\bar{B} + \bar{A}B$

bxxv) How many NOR gates are required to form an AND gate?
a. 4 b. 3 c. 2 d. 1

bxxvi) How many NOR gates are required to form an OR gate?
a. 4 b. 3 c. 2 d. 1

bxxvii) In a 2-level SOP circuit, the AND and OR gates can be replaced with only _____ gates:
a. NAND b. NOR c. NOT d. XNOR

bxxviii) In a 2-level POS circuit, the OR and AND gates can be replaced with only _____ gates:
a. NAND b. NOR c. NOT d. XNOR

SA

Q3. **Short Answer type questions:** **1 each**

i) What are the two states a Boolean variable can take?
ii) What do you mean by the term Truth Table with respect to Boolean algebra?
iii) Draw the truth table for a 2 variable Boolean AND operation.
iv) Draw the truth table for a 2 variable Boolean OR operation.
v) Draw the truth table for a Boolean NOT operation.
vi) Write any one of the Distributive Rules for a three variable Boolean expression.
vii) Write any one of the Associative Rules for a three variable Boolean expression.
viii) Write any one of the Idempotent Rules for a three variable Boolean expression.
ix) Draw the truth table for the Boolean expression f = $\bar{X} + Y$
x) Draw the truth table for the Boolean expression f = $\bar{X} + \bar{Y}$
xi) Draw the truth table for the Boolean expression f = $X + \bar{Y}$
xii) Draw the truth table for the Boolean expression f = $\bar{X} \cdot Y$
xiii) Draw the truth table for the Boolean expression f = $\bar{X} \cdot \bar{Y}$
xiv) Draw the truth table for the Boolean expression f = $X \cdot \bar{Y}$
xv) Simplify the Boolean expression f = X + X Y + Y
xvi) Simplify the Boolean expression f = X + X Y + X Y Z
xvii) Simplify the Boolean expression f = $\bar{X} + \bar{X}$ Y + X
xviii) Simplify the Boolean expression f = B $\bar{C}$ + A B $\bar{C}$
xix) Simplify the Boolean expression f = A B + A $\bar{B}$
xx) Simplify the Boolean expression f = B + A B + $\bar{B}$
xxi) Simplify the Boolean expression f = $\bar{A}$ B + $\bar{B}$
xxii) Simplify the Boolean expression f = $\bar{A}$ B + A B + $\bar{B}$
xxiii) Simplify the Boolean expression f = $\bar{A}$ B + $\bar{A}$ B + B
xxiv) Simplify the Boolean expression f = $\overline{XY}$ + XY

xxv) Simplify the Boolean expression $f = \bar{X}\,\bar{Y} + X\,\bar{Y} + \bar{Y}$

xxvi) Simplify the Boolean expression $f = P\,Q + P\,R + P\,\bar{Q}$

xxvii) Write the full min term values for the three variable min terms m_5 and m_7

xxviii) Write the full min term values for the three variable min terms m_2 and m_6

xxix) Write the full min term values for the three variable min terms m_0 and m_3

xxx) Write the full min term values for the three variable min terms m_1 and m_4

xxxi) Write the full max term values for the three variable max terms M_3 and M_5

xxxii) Write the full max term values for the three variable max terms M_1 and M_6

xxxiii) Write the full max term values for the three variable max terms M_0 and M_4

xxxiv) Write the full max term values for the three variable max terms M_2 and M_7

xxxv) Write the full SOP expression for the Boolean expression $F(X,Y,Z) = \Sigma\,(3, 6)$

xxxvi) Write the full SOP expression for the Boolean expression $F(X,Y,Z) = \Sigma\,(1, 4)$

xxxvii) Write the full SOP expression for the Boolean expression $F(X,Y,Z) = \Sigma\,(2, 7)$

xxxviii) Write the full SOP expression for the Boolean expression $F(X,Y,Z) = \Pi\,(1, 7)$

xxxix) Write the full SOP expression for the Boolean expression $F(X,Y,Z) = \Pi\,(0, 6)$

xl) Write the full SOP expression for the Boolean expression $F(X,Y,Z) = \Pi\,(2, 4)$

xli) Write the SOP expression in Σ form corresponding to the POS expression $F=\Pi\,(2, 3, 4)$

xlii) Write the SOP expression in Σ form corresponding to the POS expression $F=\Pi\,(0, 1, 5)$

xliii) Write the SOP expression in Σ form corresponding to the POS expression $F=\Pi\,(1, 3, 5)$

xliv) Write the POS expression in Π form corresponding to the SOP expression $F= \Sigma\,(2, 3, 4)$

xlv) Write the POS expression in Π form corresponding to the SOP expression $F= \Sigma\,(0, 2, 7)$

xlvi) Write the POS expression in Π form corresponding to the SOP expression $F= \Sigma\,(1, 5, 6)$

xlvii) Find the canonical SOP form of the Boolean expression $F(A,B,C) = A\bar{B} + A\bar{B}C$

xlviii) Find the canonical POS form of the Boolean expression $F(A,B,C) = (A+\bar{B})(A+\bar{B}+C)$

xlix) What is a 'don't care' term in a Boolean expression?

l) Draw the circuit symbol of a NOT gate.

li) What is the name of the complement of the XOR gate?

lii) Write the expanded output expression for an XOR operation with the inputs A and B.

liii) Write the expanded output expression for an XNOR operation with the inputs A and B.

liv) Which logic gates are called universal gates?

lv) How many NAND gates are required to design a NOT gate?

lvi) How many NOR gates are required to design a NOT gate?

lvii) How many NAND gates are required to design an OR gate?

lviii) How many NAND gates are required to design an XOR gate?

lix) How many NOR gates are required to design an AND gate?

lx) How many NOR gates are required to design an XOR gate?

lxi) How many NAND gates are required to design a NOR gate?

lxii) How many NOR gates are required to design a NAND gate?

lxiii) In a 2-level SOP circuit the AND-OR combination can be replaced by which gate combination?

lxiv) In a 2-level POS circuit the OR-AND combination can be replaced by which gate combination?

lxv) What is the value of the expression $f=1\oplus1\oplus1$?

Q4. **Long Answer type questions:** **7 each**



i) Prove De Morgan's theorems. Simplify the Boolean expression $f = \overline{(B\bar{C} + \bar{A}D)(A\bar{B} + C\bar{D})}$. 4+3

ii) Prepare truth table for the Boolean expression: $f1 = A + A\,B + \bar{B}$. Simplify the following Boolean expression using proper rules: $f2 = \bar{A}\,B\,\bar{C} + A\,B\,\bar{C} + B\,\bar{C}\,D$. Write the SOP expression for the Boolean function: $f3\ (A,B,C,D) = m_7 + m_{13}$. 4+2+1

iii) Prepare truth table for the Boolean expression: $f1 = A\,(B\,\bar{C} + \bar{B}\,C)$. Simplify the following Boolean expression using proper rules: $f2 = A\,\bar{B}\,\bar{C} + \bar{A}\,\bar{B}\,\bar{C} + \bar{A}\,B\,\bar{C} + \bar{A}\,\bar{B}\,C$. Write the SOP expression for the Boolean function: $f3\ (A,B,C,D) = m_{10} + m_{15}$. 4+2+1

iv) Prepare truth table for the Boolean expression: $f1 = \bar{A}\,\bar{B}\,(\bar{A} + B + \bar{C})$. Simplify the following Boolean expression using proper rules: $f2 = A\,B + A\,\bar{B} + \bar{A}\,C + \bar{A}\,\bar{C}$. Write the POS expression for the Boolean function: $f3\,(A,B,C,D) = M_7 \cdot M_{15}$. 4+2+1

v) Prepare truth tables for the following Boolean expression $f1 = X\,\bar{Y}\,Z + X\,(\bar{Y} + \bar{Z})$. Simplify the following Boolean expression using proper rules: $f2 = A\,B + A\,\bar{B} + \bar{A}\,C + \bar{A}\,\bar{C}$. Write the POS expression for the following Boolean function: $f3\,(A,B,C,D) = M_2 \cdot M_{13}$. 4+2+1

vi) Prepare truth tables for the following Boolean expression $f1 = X\,Y\,Z + \bar{X}\,\bar{Y}\,\bar{Z}$. Simplify the following Boolean expression using proper rules: $f2 = X\,Y + \bar{X}\,\bar{Y}\,(Y\,\bar{Z} + \bar{X}\,\bar{Y})$. Write the POS expression for the following Boolean function: $f3\,(A,B,C,D) = M_9 \cdot M_{12}$. 4+2+1

vii) Simplify the Boolean expression using proper rules: $f = AB + \bar{A}\bar{B}C + BC + \bar{B}\bar{C}A + CA + \bar{C}\bar{A}B$. Derive the expanded POS expression for $F(A,B,C,D)=\Sigma(0,2,4,8,9,10,11,12,13,15)$. 3+4

viii) Simplify the Boolean expression using proper rules: $f = A\,(A+B+C)(\bar{A}+B+C)(A+\bar{B}+C)(A+B+\bar{C})$. Derive the expanded SOP expression for $F(A,B,C,D)=\Pi(1,2,3,5,7,8,9,10,12,14)$. 3+4

ix) Simplify the Boolean expression using proper rules $f = A(B+A\bar{B}C)(B+\bar{A}\,\bar{B}\,\bar{C})(\bar{A}+\bar{B}+C)(\bar{A}+\bar{B}+\bar{C})$. Derive the expanded POS expression for $F(A,B,C,D)=\Sigma(1,2,3,4,5,6,7,10,11,12)$ 3+4

x) Simplify the Boolean expression using proper rules: $f = \overline{\overline{CD}+A} + A + \overline{CD} + AB$. Using perfect induction show the following pair of Boolean expressions are identical: A(B+C)+B+C & B+C. 3+4

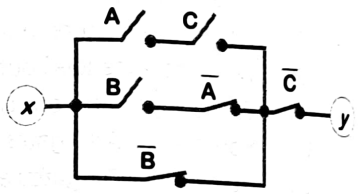
xi) Simplify the Boolean expression using proper rules: $f = \overline{(A+\bar{A}B)(B+A\bar{B})}$. Using perfect induction method show that the following Boolean expression is identical to 1: $f2 = (A+B)(A+\bar{B}C)+\bar{A}$. 3+4

xii) Simplify the Boolean expression using proper rules: $f = \overline{X\,Y + \bar{X}\,\bar{Y}\,(Y\,\bar{Z} + \bar{X}\,\bar{Y})}$. Using perfect induction show the following two Boolean expressions are identical: $(X+\bar{X}Y)(X+\bar{X}Z)$ & $X+YZ$. 3+4

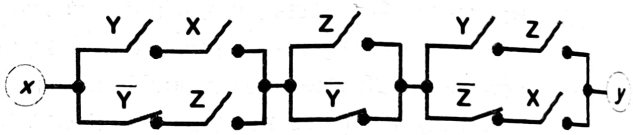
xiii) Simplify the Boolean expression using proper rules: $f = \overline{A\,(\overline{B+\bar{A}})}\,D$. Convert the following Boolean expression to canonical SOP form: F(A,B,C) = AB + BC + CA. 3+4

xiv) Simplify the Boolean expression using proper rules: $f = Y\,(YX+\bar{Y}Z+Y)\,\overline{(X\,(X+\bar{Y}))}$. Convert the following Boolean expression to canonical POS form: $F(X,Y,Z) = Y\,(\bar{X}+Z)\,(X+\bar{Y}+Z)$. 3+4

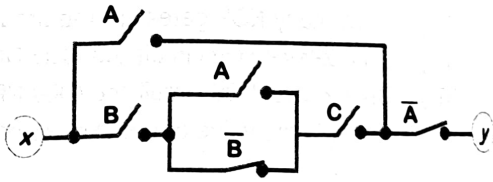
xv) Convert the following Boolean expression to canonical SOP form: $F(X,Y) = X + \bar{Y}$. Find the equivalent Boolean expression between the points ***x*** and ***y*** for the following switching circuit and simplify it: 3+4



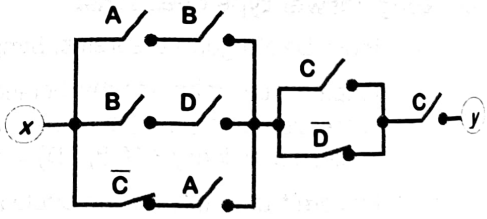
xvi) Convert the following Boolean expression to canonical POS form: $F(X,Y) = X\,\bar{Y}$. Find the equivalent Boolean expression between the points ***x*** and ***y*** for the following switching circuit and simplify it: 3+4



xvii) Simplify the Boolean expression using proper rules: $f = \overline{\bar{C}\,(\bar{B}\,\bar{C} + B\,C)}$. Find the equivalent Boolean expression between the points ***x*** and ***y*** for the following switching circuit and simplify it: 3+4



xviii) Simplify the Boolean expression using proper rules: $f = (A+C+D)\,(A+C+\bar{D})\,(A+\bar{C}+D)\,(A+\bar{B})$. Find the equivalent Boolean expression between the points ***x*** and ***y*** for the following switching circuit and simplify it: 3+4



xix) Simplify the Boolean expression using Boolean rules: $f = (A+B+C)\,(A+\bar{B}+\bar{C})\,(A+B+\bar{C})\,(A+\bar{B}+C)$. Show that $(\overline{A+B})(\overline{A+\bar{B}}) = \bar{A}\,\bar{B}$. Find the dual of the Boolean expression A+1=1. 3+3+1

xx) Simplify the Boolean expression using Boolean rules: $f = \overline{XY\,(YZ + \bar{Z}\bar{X})}$. Show that $\overline{A\,B + \bar{A}\,\bar{B}} = \bar{A}\,B + A\,\bar{B}$. Find the dual of the Boolean expression A+1=1. 3+3+1

xxi) Simplify the Boolean expression using Boolean rules: $f = \overline{[A\bar{B}\cdot A]\cdot[\bar{A}B\cdot B]}$. Convert the expression $f = \bar{X}YZ + X\bar{Y}\bar{Z} + \bar{X}Y\bar{Z}$ into POS form. Find the dual of the Boolean expression A+A=A. 3+3+1

xxii) Simplify the Boolean expression using Boolean rules: $f = (\bar{X}+XY+\bar{Y})\,(\bar{Y}+YZ+\bar{Z})\,(\bar{Z}+ZX+\bar{X})$. Convert the expression F(A,B,C) = (A+B)(B+C) into canonical SOP form. 3+4

xxiii) Simplify the Boolean expression using Boolean rules: $f = \overline{(A+B)(B+C)(A+C)}$. Convert the expression F(A,B,C) = $(\bar{A}+B+C)(B+C)$ into canonical POS form. 3+4

xxiv) Simplify the Boolean expression F(A,B,C,D) = $\Sigma(0, 1, 2, 3, 5, 8, 10, 11)$, d=$\Sigma(4, 7, 9)$, using K'map. Convert the expression F(X,Y,Z) = $X(Y+\bar{Z})(\bar{X}+Z)$ into canonical POS form. 3+4

xxv) Write the names of the universal gates. Draw the truth table of an XNOR gate. Simplify the Boolean expression F(A,B,C,D) = $\Pi(1, 2, 8, 14)$ using K'map. 2+2+3

xxvi) Draw the circuit diagram for the SOP expression $f = \bar{A}\,B + A\,\bar{B} + \bar{A}\,\bar{B}$. Show how to form an OR logic gate using only NAND gates. 4+3

xxvii) Draw the circuit diagram for the SOP expression $f = \bar{A}\,B\,\bar{C} + A\,\bar{B}\,C + B\,C$ using NAND gates only. Show how to form an AND logic gate using only NOR gates. 4+3

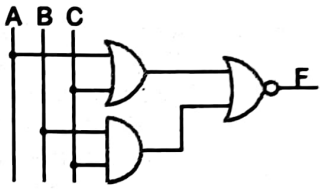
xxviii) Draw the circuit diagram for the POS expression f = $(\bar{A}+B)(A+\bar{B})(A+B)$. Show how to form a NAND logic gate using only NOR gates. 4+3

xxix) Draw the circuit diagram for the POS expression f = $(A+B+C)\,(\bar{A}+B)(A+\bar{B}+\bar{C})$ using NOR gates only. Show how to form a NOR logic gate using only NAND gates. 4+3

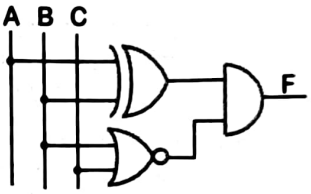
xxx) Draw the truth table for a NAND operation. Show how to form an XOR gate using only NAND gates. How many NOR gates are required to form a NAND gate? 2+4+1

xxxi) A certain 4 input gate called a ABSURD gate realises the function: ABSURD(A,B,C,D) = BC(A+D). Show a realisation of the function f(w,x,y,z) = $\Sigma(0, 1, 6, 9, 10, 11, 14, 15)$ with only three ABSURD gates and one OR gate (assume complemented inputs as available). Find the simplified POS expression for F(A,B,C,D = $\Pi(1, 2, 3, 4, 6, 9, 10, 12, 14)$ using K'map. 4+3

xxxii) Find the output F of the digital circuit shown on the right. Simplify the output expression and draw an equivalent circuit using the simplified form. State if the simplified circuit can be formed using a single gate of a given type or not. Simplify the Boolean expression F(A,B,C,D)=Σ(0,4,8,7,10,12,14,15), d=Σ(1,3,9,11), using K'map. 2+1+1+3



xxxiii) Find the output F of the digital circuit shown on the right. Simplify the output expression. Design a circuit from the simplified expression using 2-input NAND gates only. Simplify the Boolean POS expression F(A,B,C,D)=Π(1,5,11,13,15), d=Σ(9,10,14), using K'map. 2+2+3



xxxiv) A three input logic gate called EXPLODE was mass produced by a company. Experimental evidence showed that the input combinations 101 and 010 cause the gate to explode. Determine whether the gate is useless or there is a possibility to externally modify the gate (with the minimum of changes and cost) so that it can be efficiently used in other ways to implement any switching function without the scope for an explosion. The function realised by the present gate is: f=Σ(0,2,3,4). Find the simplified POS expression for Σ(0,2,4,8,9,10,11,12,13) using K'map. 4+3

xxxv) There are 4 parallel railway tracks in a place. If three or more trains pass this place, a warning signal 'S' flashes on. Also if trains simultaneously occupy the middle two tracks the signal flashes.

- 1 indicates the presence of a train on a track and 0 indicates the absence of a train on a track.
- 1 indicates that the warning signal is on and 0 indicates that the warning signal is off.

Write down the truth table for the above problem along with the output 'S'. Derive the Boolean expression from the truth table and simplify it. 4+1+2

xxxvi) You are presented with a set of requirements under which an insurance policy will be issued to an applicant by an insurance company. The applicant must be:

- A married person 25 years or over in age, or
- A married female under 25 years

The variables x, y, and z assume the truth value 1 and 0 in the following cases:

- x=1 if applicant is married, x=0 if applicant is unmarried
- y=1 if applicant is a male, y=0 if applicant is a female
- z=1 if applicant is over or equal to 25, z=0 if applicant is below 25 years age

Write down the truth table for the above problem indicating the options under which a policy can be issued. Derive the Boolean expression from the truth table and simplify it. 4+1+2

xxxvii) A library can issue up to 5 books to a member under any one of the following conditions:

- The member is 18 years or over in age, and his/her book return track record is good
- A member is below 18 years and any parent of his/her is also a member

The variables A (age), T (track record), and P (parent) assume the truth value 1 and 0 if:

- A=1 if applicant is 18 years or more, A=0 otherwise
- T=1 if applicant has good track record, T=0 otherwise
- P=1 if applicant has either of his/her parents as a member, P=0 otherwise

Write down the truth table for the above problem indicating the options under which the library can issue a book. Derive the Boolean expression from the truth table and simplify it. 4+1+2

xxxviii) An warehouse air-conditioning system is turned on if any one of the following conditions occurs:

- The weight of the stored material is less than 100 tons, the relative humidity is below 60%, but the temperature is above 30 degrees Celsius
- The weight of stored material is 100 tons or more and temperature is above 30 degrees

The following input conditions can be used:

- W=1 indicates a weight of 100 tons or more
- H=1 indicates a relative humidity of more than or equal to 60%
- T=1 indicates a temperature above 30 degrees Celsius

Draw the truth table and find an expression which assumes the value 1 whenever the air-conditioning needs to be turned ON. Find the simplified form of the above expression. 4+1+2

xxxix) A safe has 4 electronic locks – w, x, y, and z, all of which must be unlocked for the safe to open. The electronic keys to the locks, in the form of magnetic cards, are distributed amongst 4 executives in the followng manner. Mr. A has keys for locks w & x, Mr. B has keys for locks w & y, Mr. C has keys for locks z & y, and Mr. D has keys for locks x & z. The executives have to swipe their respective cards to open the safe. Determine using Boolean algebra, the names of the minimal number of executives required to open the safe.

Draw the truth table to realise the above logic. Derive the simplified Boolean expression for the same using a K'map. Draw the gate diagram to realise the electronic device which will sense the input from the cards and determine whether to unlock the safe or not. 3+2+2

xl) An airplane is to be equipped with a warning system that alerts the pilot under certain conditions of danger. The warning system monitors three instruments on the control panel. These instruments indicate the altitude, the airspeed, and the state of the landing wheels. The table below indicates specifically the relationship of these inputs to logic states.

Two conditions of danger must be detected. First, a "Landing Warning" lamp glows if airspeed is less than cruising speed and landing wheels are not down, or if wheels are down and the pilot is at proper landing altitude but not at proper landing speed. Second, an "Airframe Warning" lamp glows whenever the airspeed is too fast, or when wheels are down and airspeed is above landing speed. Also, when either lamp glows, an alarm buzzer should sound to get the pilot's attention. Assume that logic level voltages (high=1, low=0) are generated by the warning system. Draw a truth table showing the inputs and the three outputs. Find the simplified output for the alarm buzzer. 5+2

Altitude	**Wheels**	**Airspeed**
'0' Means Landing Altitude '1' Means Cruising Altitude	'0' Means Wheels Up '1' Means Wheels Down	'00' Means Too Slow '01' Means Landing Speed '10' Means Cruising Speed '11' Means Too Fast ↓ Speed Increasing

CHAPTER 5
Combinational Circuits

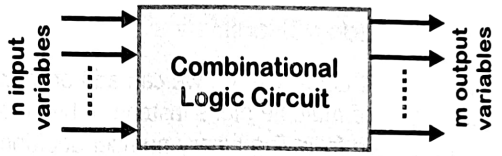
- General Description of Combinational Circuits 5-1
- Adder Circuits 5-1
- Subtractor Circuits 5-3
- Multiple-bit Adder and Subtractor Circuits 5-5
- Multiplexer Circuit 5-7
- De-multiplexer Circuit 5-8
- Decoder Circuit 5-9
- Encoder Circuits 5-10

During the 1930's the logic circuits of computers were constructed from mechanical switches and relays. These were comparatively slow, large, and produced a great amount of heat.

5.1 General Description of Combinational Circuits

When the **output of a digital circuit depends on the present state of inputs only** and not on the previous set of inputs, then that circuit is known as a combinational circuit. The different digital circuits that we discussed in the previous chapter are examples of combinational circuits. The output of a combinational circuit can be fully determined by a set of Boolean functions and designed using logic gates.

The basic constituents of a combinational circuit are a set of input variables, logic gates to implement the logic, and a set of output variables. The inputs are applied to the logic gates and the processed variables are available at the output as output variables. The input and the output are in the form of binary numbers or signals. The figure above shows a schematic diagram of a combinational circuit with 'n' number of input variables and 'm' number of output variables. Let us now discuss the working principle of some important combinational circuits that are used in a computer.



Combinational circuit block diagram

5.2 Adder Circuits

The basic function of the Arithmetic and Logic Unit (ALU) involves arithmetic operations. Of this the **most important and basic arithmetic operation** is to **add two binary digits**. To achieve this, the logical circuit that is used is known as a **Half Adder circuit**. The name Half Adder implies doing half of the add operation as no option is there for adding any carry-in bit. **George Stibitz** built the first adder around 1937.



• Half Adder Circuit:

The basic operation of an adder circuit is to **add two binary digits**. There are four different possibilities as (0+0), (0+1), (1+0), and (1+1). The result of addition is shown below for the two bits **X** and **Y**. We find that except for the last combination, the first three combinations are same as decimal addition. For the last combination we know that (1+1)=2 in decimal addition and the binary equivalent of 2 is $(10)_2$ in base 2. For the result 10_2, the digit under unit's place is '0' and '1' is carried forward to ten's place.

X →	0	0	1	1
Y →	+ 0	+ 1	+ 0	+ 1
Sum →	0	1	1	1 0
				↗ Carry

The truth table for adding two binary digits is shown on the right. In the table, X and Y represent the two binary digits, **S represents the sum** and **C represents any carry forward**. We can see that the sum is 0 when both the digits are zero or when both the digits are 1. **The sum bit is 1 only when either of the input digits is 1.** The carry output is 1 only when both the inputs are 1.

X	Y	Sum (S)	Carry (C)	Sum min terms	Carry min terms
0	0	0	0		
0	1	1	0	$\bar{X}Y$	
1	0	1	0	$X\bar{Y}$	
1	1	0	1		XY

Sum: Half Adder $S = \bar{X}Y + X\bar{Y}$

Carry: Half Adder $C = X.Y$

Truth table of Half Adder

From the truth table we can write the **SOP expression** for the sum and the carry as:

$$\text{Sum, } S = \bar{X}Y + X\bar{Y} = X \oplus Y \text{ and Carry } C = XY$$

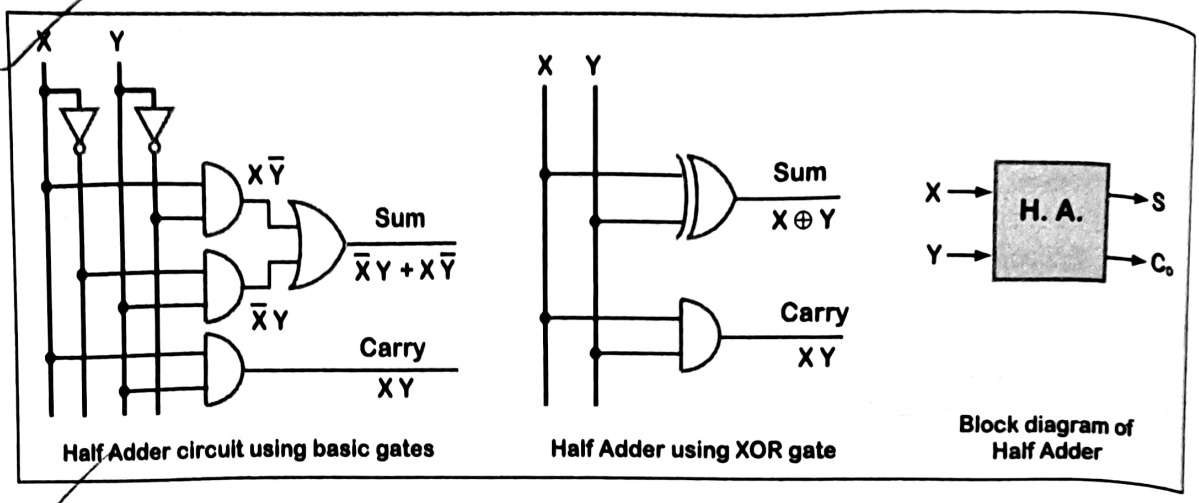
Expression for Sum & Carry of Half Adder

By a closer inspection we can see that the **sum expression of a half adder** is basically an **XOR operation.** Thus we can replace the sum expression by an equivalent XOR expression as $S = X \oplus Y$.

Circuit diagrams of a half adder, using basic gates and using a XOR gate are given below. When multiple half adders are used in a circuit, to minimise complexity, the gate circuit is sometimes replaced by a block diagram as shown. The internal circuit is represented by a box, with input digits **X** and **Y**. The output is taken as **S** for the sum, and C_o for the carry out.

Circuits for Half Adder

Half Adder can be constructed using XOR gate and AND gate



- **Full Adder Circuit:**

Using a half adder circuit, we can add only two binary digits. However if we want to add two binary numbers consisting of multiple digits instead of two digits, we have to include a third bit to account for the carry input. Consider the following binary number additions:

1	1	1	1	⇐ Carry-In (Z)
1 1	1 1	0 1	0 1	⇐ X
+ 0 1	+ 1 1	+ 1 1	+ 0 1	⇐ Y
1 0 0	1 1 0	1 0 0	1 0	⇐ Sum

From the first addition we can see that for the ten's place (circled column) we have to add (1+0)+1 (carry) i.e. we have to **add three digits**. The remaining three additions show some of the different possibilities that can arise for the Ten's place for different values of the digits X and Y. To carry out the function of adding 3 binary digits, a **Full Adder circuit** is used.

Full Adder Truth Table

The expression for Sum in case of an Adder cannot be simplified further

The truth table for a full adder circuit is given on the right. In the table, **X** and **Y** represent the binary digits to be added, and **Z=C_{in}** represents the Carry-in bit of the result of addition from the previous column. **S** represents the Sum, and C_o represents the Carry-out bit generated by adding the digits X and Y. Therefore we have three input columns X, Y, Z, and **two output columns S and C_o**. The min terms are also written.

X	Y	Z	Sum (S)	Carry (C_o)	Sum min terms	Carry min terms
0	0	0	0	0		
0	0	1	1	0	$\bar{X}\bar{Y}Z$	
0	1	0	1	0	$\bar{X}Y\bar{Z}$	
0	1	1	0	1		$\bar{X}YZ$
1	0	0	1	0	$X\bar{Y}\bar{Z}$	
1	0	1	0	1		$X\bar{Y}Z$
1	1	0	0	1		
1	1	1	1	1	XYZ	XYZ

From the truth table we can write the SOP expressions for **Sum** and **Carry** as:

Expression for Sum in F.A.

$$S = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ = \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + \bar{X}\bar{Y}Z + XYZ$$
$$= (\bar{X}Y + X\bar{Y})\bar{Z} + (\bar{X}\bar{Y} + XY)Z = (X \oplus Y)\bar{Z} + (\overline{X \oplus Y})Z = (X \oplus Y) \oplus Z = X \oplus Y \oplus C_{in}$$

Note that the expression $(\bar{X}\bar{Y} + XY)$ is equal to the complement $\overline{\bar{X}Y + X\bar{Y}}$ i.e. equal to $(\overline{X \oplus Y})$. The interested student can simplify the complement and check the result. Also remember that a **binary sum operation is always an XOR operation** for any number of bits. Let us now simplify C_o.

Expression for Carry in F.A.

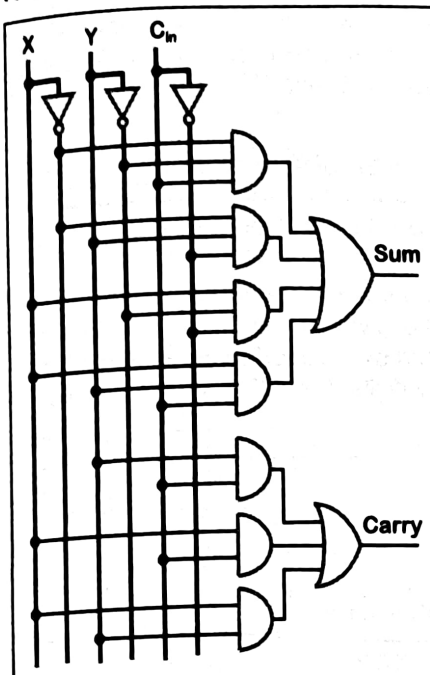
$$C_o = \bar{X}YZ + X\bar{Y}Z + XY\bar{Z} + XYZ = \bar{X}YZ + XYZ + X\bar{Y}Z + XYZ + XY\bar{Z} + XYZ \quad [\text{As } A+A=A]$$
$$= (\bar{X} + X)YZ + (\bar{Y} + Y)XZ + (\bar{Z} + Z)XY = YZ + XZ + XY \quad [\text{As } (\bar{A}+A)=1]$$
$$= XY + XC_{in} + YC_{in}$$

An **alternative expression** for the carry out C_0 can also be derived as:

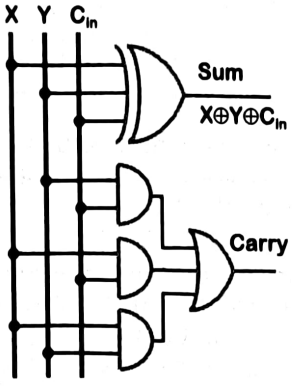
$C_0 = \bar{X}YZ + X\bar{Y}Z + XY\bar{Z} + XYZ = (\bar{X}Y + X\bar{Y})Z + XY(\bar{Z} + Z) = (X \oplus Y)Z + XY$
$= (X \oplus Y)C_{in} + XY$

Alternative expression for F.A. Carry Out

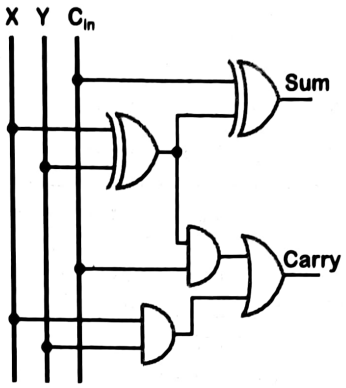
Various **gate diagrams** of a **Full Adder** are given below. Any one of these can be used.



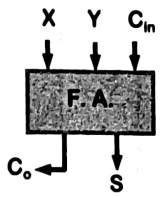
Full Adder Circuit using basic gates



Full Adder using XOR gate



Full Adder using 2 input XOR gate and alternative representation for C_0



Block diagram of Full Adder

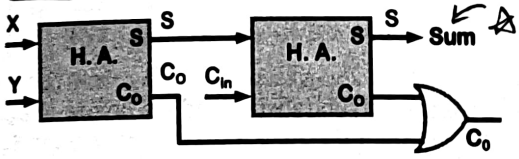
Various representations of a **FULL ADDER** circuit

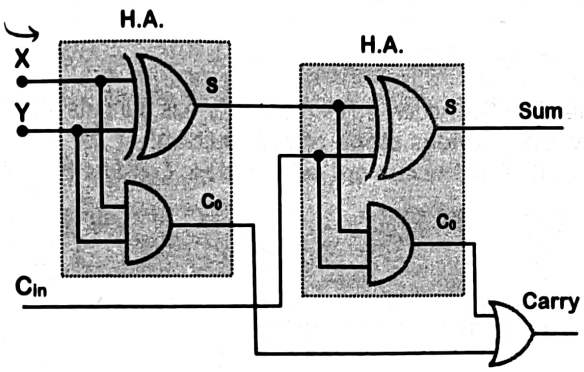
Full Adder Sum: $X \oplus Y \oplus C_{in}$

Full Adder Carry: $(X \oplus Y)C_{in} + XY$ OR $XY + XC_{in} + YC_{in}$

- **Full Adder Using Half Adder Circuits:**

One can also design a Full Adder circuit using Half Adder circuits. This requires **two Half Adder circuits and an OR gate.** The first Half Adder adds the two binary bits X and Y. The second Half Adder is then used to add the sum X⊕Y obtained from the first Half Adder, and the carry-in bit C_{in}.





Full Adder designed using Half Adders

The sum output from the second Half Adder (H.A.) gives the final sum of adding the three bits X, Y, and C_{in}. The **alternative expression for C_0** is used to get the final carry-out by combining the carry-out from the two half adders using an OR gate. The **block diagram** for such a design along with the **gate diagram** is shown (note that the inputs for the second H.A. are **X⊕Y** from the first H.A., and **C_{in}**).

Two Half Adder circuits can be used to design a Full Adder

5.3 Subtractor Circuits

Just like adding two binary digits, they can be subtracted also. A **Half Subtractor** circuit is used to **subtract two binary digits.** In carrying out subtraction we have the **difference** and the **borrow** outputs (instead of the sum and carry outputs for a half adder).

- **Half Subtractor Circuit:**

The subtraction of two binary digits gives four different cases viz. (0–0), (0–1), (1–0), and (1–1). The result of subtraction is shown on the right for the bits **X** and **Y**.

X →	0	1 0	1	1
Y →	− 0	− 1	− 0	− 1
Diff. →	0	1	1	0
		Borrow		

Half Subtractor Logic

Truth Table Half Subtractor

X	Y	Diff. (D)	Borrow (B)	Difference min terms	Borrow min terms
0	0	0	0		
0	1	**1**	**1**	$\bar{X}Y$	$\bar{X}Y$
1	0	**1**	0	$X\bar{Y}$	
1	1	0	0		

We find that except for the second combination, the remaining three combinations are same as decimal subtraction. For the **second combination**, we know that to subtract 1 from 0 (i.e. to do 0 – 1), we have to **borrow a '1'** from the next column for the digit X. When we borrow a '1', X becomes 10_2 which is same as 2_{10}. Therefore (X-Y) is same as (2-1) and the result i.e. the **Difference** is 1. Thus the digit under the Unit's place is '1' and '1' is borrowed from the next column. The truth table for binary subtraction is given on the left.

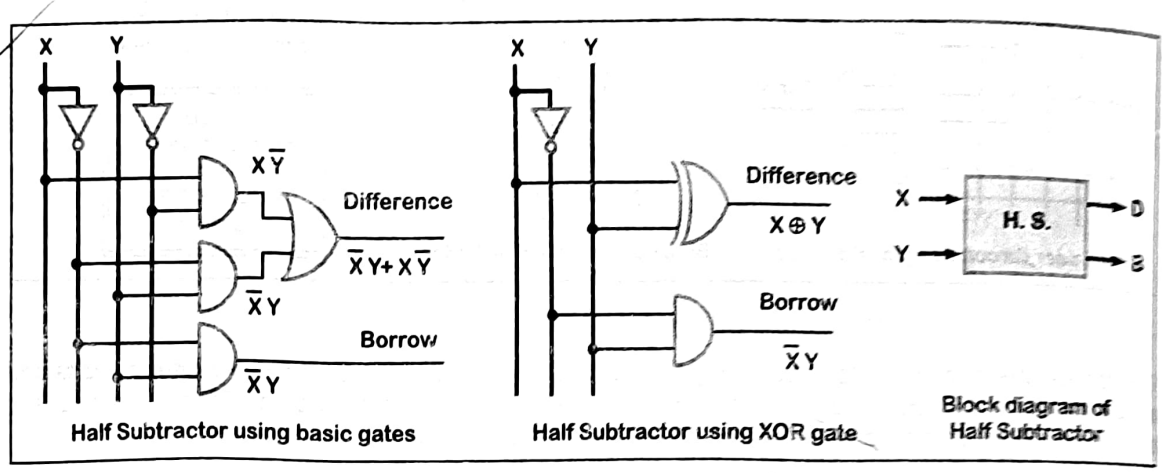
From the truth table we get the **difference D** and **borrow B** as given by the **SOP expressions**:

Difference **D** = $\bar{X}Y + X\bar{Y}$ and Borrow **B** = $\bar{X}Y$ or Difference **D** = $X \oplus Y$ and the Borrow **B** = $\bar{X}Y$

Difference & Borrow of Half Subtractor

As we see from the above expression, the **difference operation is also a XOR operation** similar to a half adder. The different circuit representations for a half-subtractor are given below. When **multiple half subtractors** are used in a circuit, the gate circuit is sometimes replaced by a block diagram as shown below. The internal circuitry is represented by a box, with inputs as the binary digits **X** and **Y**. The output is taken as **D** for **difference** bit, and **B** for the **borrow** bit.

Different circuits for Half Subtractor



Diff. Half Sub. $D = \bar{X}Y + X\bar{Y}$

Borrow Half Sub. $B = \bar{X}Y$

Result of Addition and Subtraction are both XOR operations

- Full Subtractor Circuit:

Similar to a Full Adder, a Full Subtractor has 3 inputs viz. **X** (minuend), **Y** (subtrahend) and the borrow-in B_{in} **from the previous stage of subtraction**. At the output we have the difference **D**, and a borrow bit B_o **generated by the current subtraction**. Consider the following binary subtractions:

B_o=1 B_{in}=0	B_o=1 B_{in}=1	B_o=0 B_{in}=1	B_o=0 B_{in}=0	
1 0 1	1 0 0	1 1 0	1 1 0	⇐ Borrow B
– 1 1	– 1 1	– 0 1	– 1 0	⇐ X
0 1 0	0 0 1	1 0 1	1 0 0	⇐ Y
				⇐ Difference D

Truth Table for Full Subtractor

From the first subtraction we see that for the ten's place (circled column) we have to subtract (0–1). To do this we have to borrow a 1 from the next column. This serves as the B_o bit for the current subtraction. Since no bit is taken during the subtraction of the previous column, the B_{in} bit is 0. The remaining three subtractions give the different possibilities of **D** and B_o that can arise for the ten's place for different values of **X, Y,** and B_{in}. The truth table for a full subtractor is given on the right with **Z**=B_{in}.

X	Y	Z	Diff. (D)	Borr. (B_o)	Difference min terms	Borrow min terms
0	0	0	0	0		
0	0	1	**1**	**1**	$\bar{X}\bar{Y}Z$	$\bar{X}\bar{Y}Z$
0	1	0	**1**	**1**	$\bar{X}Y\bar{Z}$	$\bar{X}Y\bar{Z}$
0	1	1	0	**1**		$\bar{X}YZ$
1	0	0	**1**	0	$X\bar{Y}\bar{Z}$	
1	0	1	0	0		
1	1	0	0	0		
1	1	1	**1**	**1**	XYZ	XYZ

$D = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ = \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + \bar{X}\bar{Y}Z + XYZ$

$= (\bar{X}Y + X\bar{Y})\bar{Z} + (\bar{X}\bar{Y} + XY)Z = (X \oplus Y)\bar{Z} + (\overline{X \oplus Y})Z = (X \oplus Y) \oplus Z$

$= X \oplus Y \oplus B_{in}$

Full Subtractor Difference & Borrow

$B_o = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + \bar{X}YZ + XYZ = \bar{X}\bar{Y}Z + \bar{X}YZ + \bar{X}Y\bar{Z} + \bar{X}YZ + XYZ + \bar{X}YZ$ [As A+A=A]

$= (\bar{Y}+Y)\bar{X}Z + (\bar{Z}+Z)\bar{X}Y + (X+\bar{X})YZ = \bar{X}Z + \bar{X}Y + YZ$ [As $(\bar{A}+A)=1$]

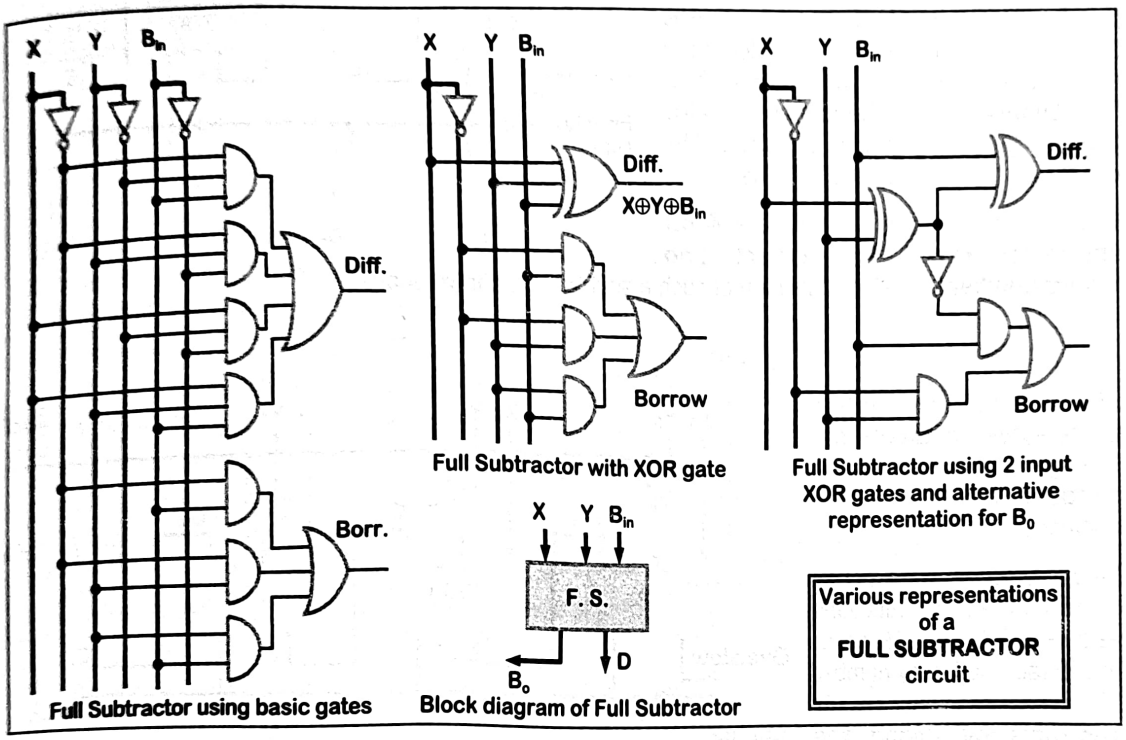
$= \bar{X}B_{in} + \bar{X}Y + YB_{in}$

Alternative expression for borrow B_o:

$B_o = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + \bar{X}YZ + XYZ = (\bar{X}\bar{Y} + XY)Z + \bar{X}Y(\bar{Z} + Z) = (\overline{X \oplus Y})Z + \bar{X}Y$

$= (\overline{X \oplus Y})B_{in} + \bar{X}Y$

Alternative expression for F.S. Borrow

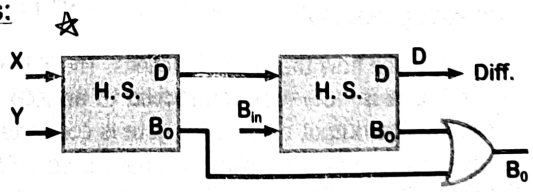


Full Subtractor Difference
$X \oplus Y \oplus B_{in}$

Full Subtractor Borrow:
$\bar{X}B_{in} + \bar{X}Y + YB_{in}$
OR
$(\overline{X \oplus Y})B_{in} + \bar{X}Y$

• Full Subtractor Using Half Subtractor Circuits:

A **Full Subtractor** can also be designed using Half **Subtractors.** It requires two half subtractors and an OR gate. The arrangement is shown on the right. The final difference is obtained as $X \oplus Y \oplus B_{in}$ at the D output of the second Half Subtractor. The final borrow-out is obtained by using the **alternative expression for B_o** at the output of the OR gate.



F.S. using Half Subtractors

5.4 Multiple-bit Adder and Subtractor Circuits

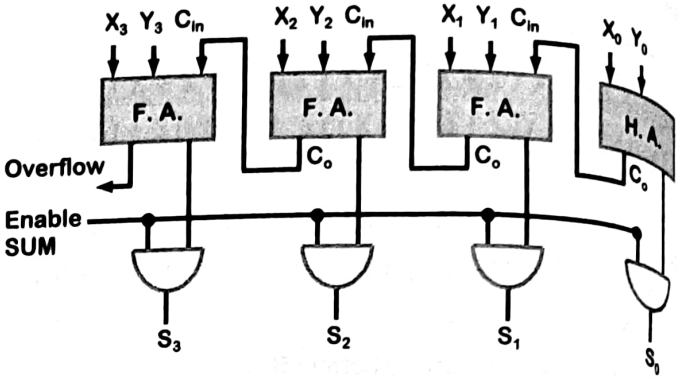
In general we add or subtract two numbers and not two binary digits only. **To add two binary numbers** we have to **use an adder circuit for every pair of bits** in the number. For example consider two 4-bit binary numbers $X_3X_2X_1X_0$ and $Y_3Y_2Y_1Y_0$. To add these two numbers we have to add each pair of bits i.e. X_0 & Y_0, X_1 & Y_1, X_2 & Y_2, and X_3 & Y_3 as shown below. The **carry from each column is fed to the next column,** where it is added as the carry-in bit along with the X and Y bits.

Multiple bit Adders

$$\begin{array}{cccc} C_{in} & C_{in} & C_{in} & \\ X_3 & X_2 & X_1 & X_0 \\ + \; Y_3 & Y_2 & Y_1 & Y_0 \\ \hline S_3 & S_2 & S_1 & S_0 \end{array}$$

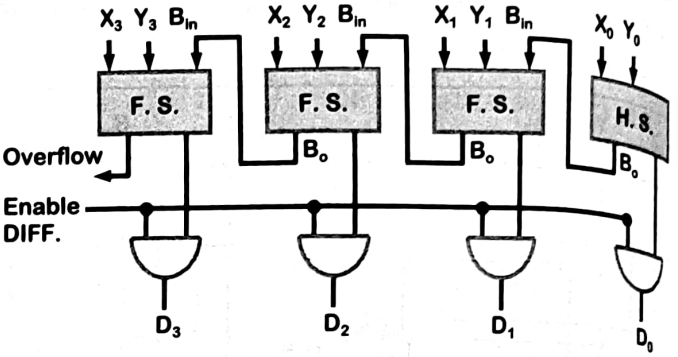
4-bit Adder Circuit

The circuit comprises of **one Half Adder** and **three Full Adders** as shown in the circuit diagram on the right. The adder circuits are shown as block diagrams. As there is no carry bit for the rightmost column, the first circuit used is a half adder. The rest three are full adder circuits with carry in bits from the previous columns.



The first HA block is used to add the two bits X_0 & Y_0. The next block is a FA as the carry from the previous HA is to be added now along with the inputs X_1 & Y_1. In this manner the four bits are added one after the other using the FA blocks. The C_0 of one block is the C_{in} of the next block. The final sum is $S_3S_2S_1S_0$. Each sum bit from an adder is fed to an AND gate. After the inputs are ready, a high **Enable** signal activates the four AND gates to get the sum bits at output. The overflow is the C_0 of the last full adder.

4-bit Subtractor Circuit



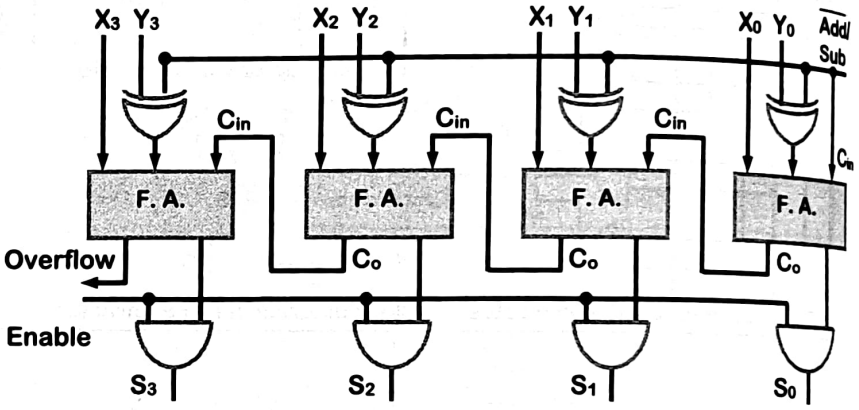
In a similar manner we can have a **4 bit subtractor circuit to subtract two binary numbers**. The block diagram of such a subtractor is shown above.

4-bit Adder-Subtractor Circuit

- <u>4 bit Adder-Subtractor Circuit:</u>

We can combine the function of the above two circuits into a single circuit to **add or subtract** two 4-bit binary numbers.

When adding two binary numbers, the circuit functions as a 4-bit full adder. However to subtract the binary number Y from X, the **circuit subtracts by taking the 2's complement of Y** and then adding that to X.



The same circuit can be used for doing Addition and 2's Complement Subtraction by adjusting the Add/Sub input.

Let us analyse how the circuit functions. The X input is directly given to the Full Adder (FA) blocks as before. The **Y input is however connected to an XOR gate**. All the XOR gates have another input from the line $\overline{\text{Add}}$/Sub. The output of the XOR gate is connected to the other input of the FA block. Note that instead of using a HA for the unit's place, we have used a FA block and the $\overline{\text{Add}}$/Sub line is applied to the $\mathbf{C_{in}}$ input of the first FA block. The reason for this is explained below.

During Addition, the $\overline{\text{Add}}$/Sub line is made low i.e. made **equal to 0**. We can see from the XOR table that whenever the **a** input is 0, the XOR output is same as the input **b**. Thus when $\overline{\text{Add}}$/Sub is '0', the output of the XOR gates are the same as the respective **Y** inputs. Moreover the $\mathbf{C_{in}}$ input to the first FA is also '0' and has no effect on the circuit. Under these conditions, the circuit functions as a simple 4-bit Full Adder circuit as before.

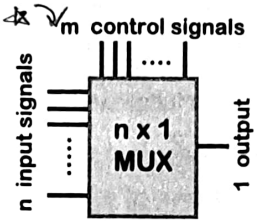
a	b	a⊕b
0	0	0
0	1	1
1	0	1
1	1	0

During Subtraction, the $\overline{\text{Add}}$/Sub line is made equal to '1'. We can see from the XOR table, that whenever **a** is high i.e. '1', the XOR output is the complement of the input **b**. Thus we get the 1's complement of the input **Y** at the output of the XOR gate, during subtraction. This output is then given to the FA blocks. Moreover the high $\overline{\text{Add}}$/Sub signal is also given to the $\mathbf{C_{in}}$ input of the first FA block. In doing so we are taking the 1's complement of the number **Y** to be subtracted and adding 1 to the LSB of that number. This is exactly how we subtract a number using 2's complement method. Thus when $\overline{\text{Add}}$/Sub is made high, **the circuit functions as a 2's complement subtractor** circuit.

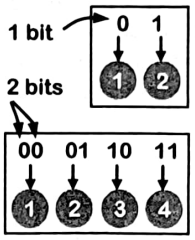
5.5 Multiplexer Circuit

Multiplexer Circuit

A multiplexer means **many-into-one**. It is a circuit with many inputs but only one output. The block diagram on the right shows an **nx1 multiplexer** (in short also called a MUX), where nx1 indicates there are 'n' number of inputs and '1' output. Apart from the '**n**' number of **input signals**, there is another set of signals called **control signals**, which determine which input signal will be transmitted to the output. Depending upon a particular combination of the **m** control signals, **only one** of the **n** input signals is selected and transmitted to the output of the multiplexer.



The number of **control bits also determine how many input signals can be handled** by the multiplexer. The general rule is that, with **m** control signals a maximum of $n=2^m$ input signals can be selected:



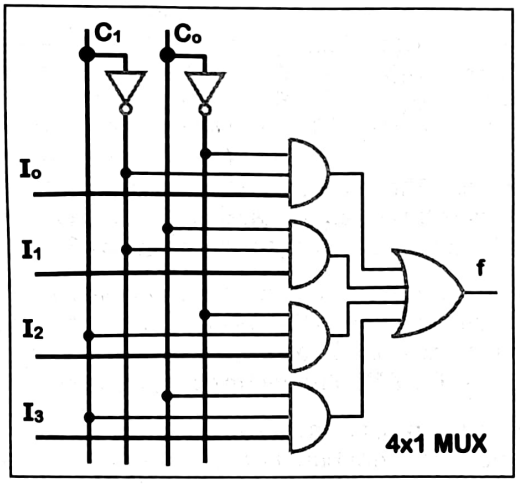
A Multiplexer circuit can select only one input from a set of inputs

This is evident from the fact that:

- 1 bit with 2 values (0 & 1) can be used to select 2^1 i.e. 2 number of inputs
- 2 bits with 4 values (00, 01, 10, 11) can be used to select 2^2 i.e. 4 no. of inputs
- Similarly **m** bits with m values can be used to select 2^m number of inputs

The next figure shows a **multiplexer with 4 inputs and 1 output**. For 4 inputs there should be 2 control signals (as $4 = 2^2$) to select any one of the inputs I_0, I_1, I_2, or I_3. C_0 and C_1 are the 2 control signals.

4x1 Multiplexer circuit diagram



The control signals C_0 and C_1 are connected to the **4 AND gates** such that, if we ignore the inputs I_0 to I_3:

- AND **gate for I_0** will be **ACTIVE** if C_1=**0**, and C_0=**0**
- AND **gate for I_1** will be **ACTIVE** if C_1=**0**, and C_0=**1**
- AND **gate for I_2** will be **ACTIVE** if C_1=**1**, and C_0=**0**
- AND **gate for I_3** will be **ACTIVE** if C_1=**1**, and C_0=**1**

To do this, for the C_0 line connect the AND gates to C_0 **alternately** between $\bar{C}_0$ and C_0 lines, starting from the NOT line. For the C_1 line, connect **two consecutive AND gates alternately** between $\bar{C}_1$ and C_1 starting from the NOT line.

In a Multiplexer, m control signals can select at the most $n = 2^m$ number of inputs

To understand the logic of the circuit refer to the diagrams shown below. In figure 1, C_1=**0** and C_0=**0**. For this combination, AND **Gate-0** will be **ON** as the control inputs to Gate-0 will consist of inverted control input from C_1, which is $\bar{0} = 1$ and the inverted control input from C_0, which is also $\bar{0} = 1$. Thus both control inputs to Gate-0 being 1, **Gate-0 should be ON**. Under such situation, a closer inspection will reveal that **all other AND gates will be OFF** since at least one of the inputs to them is '0'. The input I_0 connected to the AND Gate-0 will then be transferred to the output OR gate as is indicated by the shaded gates and the shaded line.

Similarly for the other combinations of C_1 and C_0 only one AND gate will be ON at a time as indicated by the truth table in the next page. Thus **each gate is ON only for a particular combination of C_1 & C_0** and the input signal connected to that gate only, will pass out of the OR gate.

Working of 4x1 multiplexer

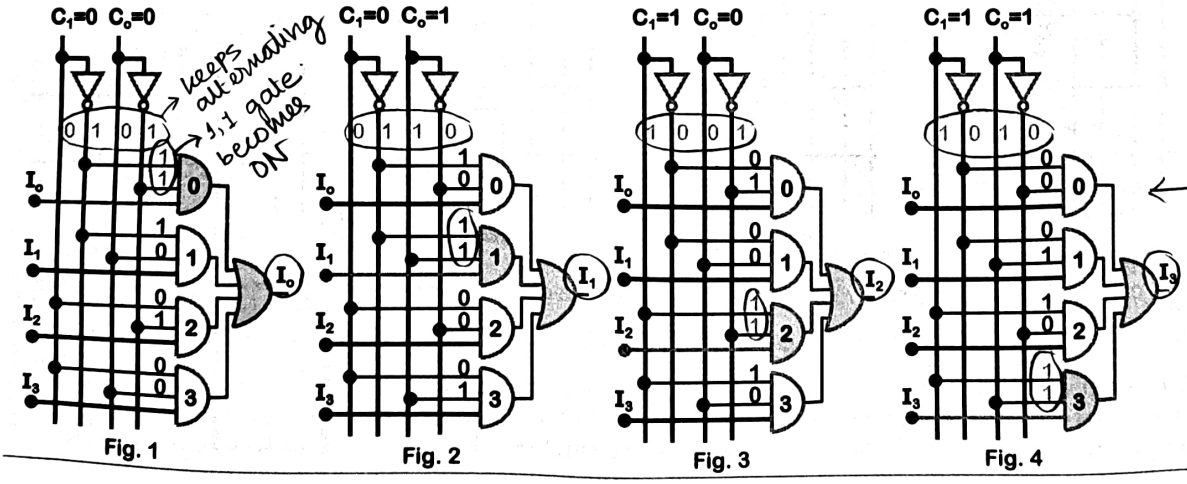


Fig. 1 Fig. 2 Fig. 3 Fig. 4

Remember combs next page

Output Table of Multiplexer

The MUX selects the particular input-data for which the AND gate is ON and transmits it to the output OR gate. Finally the **OR gate combines the output of the AND gates** and makes the final output high whenever **any one** of the AND gates has a high output. The output F of a 4x1 Multiplexer can be written as:

$$F = \overline{C_1}\,\overline{C_0}\,I_0 + \overline{C_1}\,C_0\,I_1 + C_1\,\overline{C_0}\,I_2 + C_1\,C_0\,I_3$$

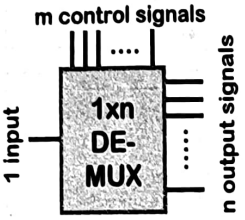
C_1	C_0	AND Gate ON	Data Selected
0	0	0	I_0
0	1	1	I_1
1	0	2	I_2
1	1	3	I_3

- **Uses of a Multiplexer:**

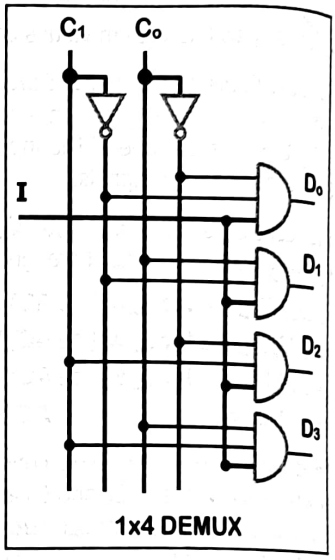
Multiplexers have various uses in digital circuits. These are used in a common bus system where the inputs from various input devices are multiplexed and transferred to the processor line. An AND-OR SOP circuit can also be made using a multiplexer. It is used in conjunction with a de-multiplexer circuit to connect communication devices over long distances.

5.6 De-multiplexer Circuit

De- Multiplexer circuit



A **De-multiplexer means one-to-many**. It is a circuit which has **one input and many outputs**. The signal on the input line is connected to all the output lines. By applying a control signal, one can select the output line to which the input signal will be transmitted. The block diagram on the left shows a **1xn DEMUX** with 'n' outputs and 'm' controls signals. The number of **control bits also determines how many output lines can be handled** by the de-multiplexer. The general rule is with **m** control signals a maximum of $n=2^m$ outputs can be selected.



1x4 DEMUX

A De-Multiplexer circuit can select only one output line from a set of outputs

In De-multiplexer m control signals can select at the most $n = 2^m$ number of output

The circuit drawn on the right shows a 1x4 DEMUX. Since the number of outputs is $2^2=4$, the number of control signals required is 2, as represented by C_1 and C_0. The input **I** is given to all the output AND gates. The **AND connections** are made **similar to a MUX**.

Depending upon the combinations of C_1 and C_0 **any one of the AND gates gets enabled at a time**. Suppose when $(C_1,C_0) \equiv (1,0)$ then AND Gate-2 will be enabled as both of its control inputs will be 1. The input signal '**I**' gets transmitted through **Gate-2** and will be available at the output terminal $\mathbf{D_2}$. Similarly for the other combinations of C_1 and C_0, the input **I** will get transmitted through that gate only which gets enabled as shown in the diagrams below:

Working of De- Multiplexer circuit

Same as 4x1 MUX

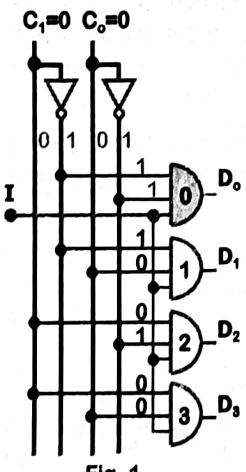


Fig. 1

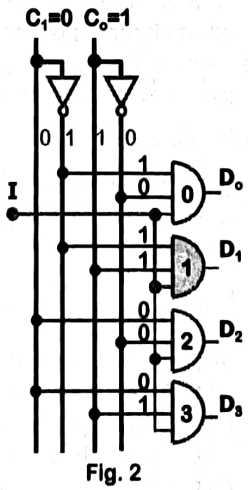


Fig. 2

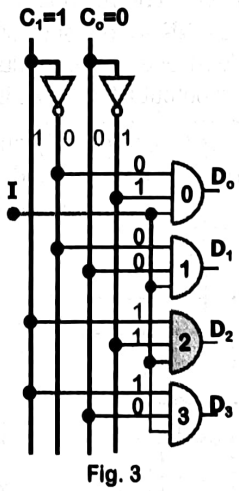


Fig. 3

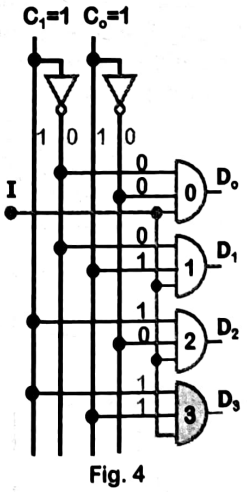


Fig. 4

Output of De- Multiplexer circuit

The output expressions for the four outputs $D_0\,D_1\,D_2\,D_3$ are given below:

$$D_0 = \overline{C_1}\,\overline{C_0}\,I, \quad D_1 = \overline{C_1}\,C_0\,I, \quad D_2 = C_1\,\overline{C_0}\,I, \quad D_3 = C_1\,C_0\,I$$

C_1	C_0	D_3	D_2	D_1	D_0
0	0	0	0	0	I
0	1	0	0	I	0
1	0	0	I	0	0
1	1	I	0	0	0

The function of a de-multiplexer is opposite to that of a multiplexer i.e. it is mainly **used to transfer information coming from a single source to different locations** by selecting the locations using control signals.

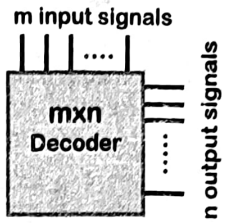
- **Uses of a De-Multiplexer:**

A de-multiplexer can also **form a common bus system**, where the data from one bus is directed to two or more different bus systems. For example the output from the processor can be transferred to different output devices like the VDU, or printer, or speaker using a de-multiplexer circuit. It is used in conjunction with a multiplexer circuit to connect communication devices over long distances.

5.7 Decoder Circuit

Decoder Circuit

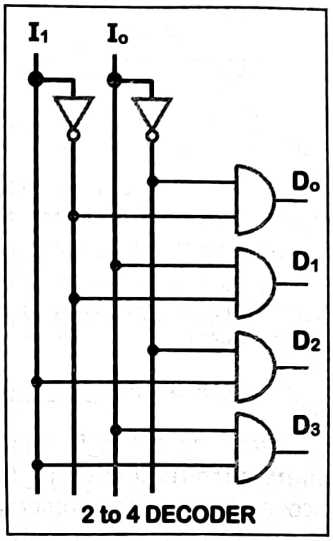
A decoder circuit is used to decode binary data and is **similar to a de-multiplexer**, with the exception that there are **no separate data input lines**. Thus if the input line of a de-multiplexer is made equal to 1, then a de-multiplexer can be used as a decoder. The **control signals** of a de-multiplexer serve as the **only inputs** to a decoder. Like a de-multiplexer, **m** control signals can be used to activate any one of the $\mathbf{n=2^m}$ outputs. The schematic diagram of a **mxn decoder** is shown on the right. Here m is the number of inputs and n the number of outputs.



I_1	I_0	D_3	D_2	D_1	D_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

2 to 4 Decoder Circuit

The circuit on the right shows the circuit diagram of a **2 to 4 Decoder**. Depending upon the input signal combination (the control signal itself is the input here) **any one of the outputs D_0, D_1, D_2 or D_3 will be high at a time.** The other outputs will be 0. When for example $(I_0,I_1) = (1,1)$ then D_3 will be high and all other outputs will be low. Similarly for each of the other combinations of I_1 and I_0, a particular output line will be high at a time. The truth table above shows the relationship between the input and the output signals. The outputs of the decoder are given by:



$D_0 = \bar{I}_1\ \bar{I}_0$, $D_1 = \bar{I}_1\ I_0$, $D_2 = I_1\ \bar{I}_0$, $D_3 = I_1\ I_0$

A Decoder circuit is used to decode a binary number and activate specific outputs

- **Uses of a Decoder:**

A Decoder can be used to decode a binary data to a different number system or to activate a circuit based on the decoded input signal.

Uses of Decoders

Binary to Decimal Decoder: A **Binary to Decimal Decoder** is used to get the **decimal digit** corresponding to a **particular** input **binary combination**. The circuit has 10 outputs corresponding to the decimal digits from 0 to 9. In such a circuit, for a particular binary combination of the input signals (I_3,I_2,I_1,I_0) the **output that is high** corresponds to the **decimal equivalent of the input** combination. Thus for input signal $(I_3I_2I_1I_0) \equiv (0011)$ the output D_3 is high which is the decimal equivalent of $(0011)_2$ [Note $11_2 = 3_{10}$].

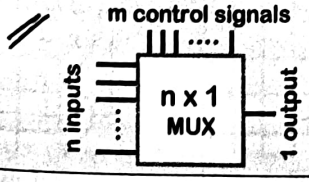
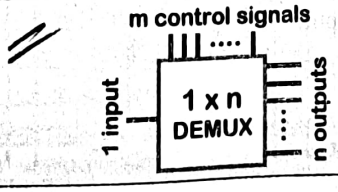
Address Decoder: Amongst its many uses, a decoder is used **to decode the address location of memory registers**. The combination of address bits on the address bus is decoded by a decoder and the decoder outputs are used to enable particular memory locations.

A Decoder circuit is used inside the Control Unit of a CPU

Instruction Decoder: A decoder is also **used in the Control Unit of the CPU to decode the program instructions** and activate particular control lines to carry out different operations in the ALU.

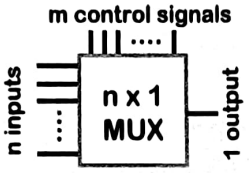
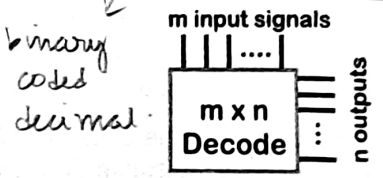
Comparison between a Multiplexer and a De-multiplexer

Comparison b/w multiplexer and de-multiplexer

Multiplexer	De-multiplexer
1. A Multiplexer has many inputs and one output	1. A De-multiplexer has one input and many outputs
2. A Multiplexer is used to select one input from many inputs	2. A De-multiplexer is used to select one output from many outputs
3. m control signals can select maximum of 2^m inputs	3. m control signals can select a maximum of 2^m outputs
4. It is used in a Common Bus System	4. It is used in a Common Bus System
	

Difference MUX & Decoder

Difference between a Multiplexer and a Decoder

Multiplexer	Decoder
1. A Multiplexer selects one input from many inputs and transmits it through the output line	1. A Decoder is used to decode a binary input
2. If there are **n** inputs, there is a single output	2. If there are **m** inputs, then there are 2^m outputs
3. Control signals are used to select a particular input	3. Control signals serve as binary inputs
4. It is used In a Common Bus System	4. It is used in a BCD to Decimal Decoder
	

5.8 Encoder Circuits

An encoder is used to **convert or encode an active input signal to a binary coded output signal**. The input signal can also be an analogue one like a variable voltage or a rotating shaft. Thus functionally it is **opposite to that of a Decoder**.

Decimal to Binary Encoder

- Decimal to Binary Encoder:

The circuit on the right shows a **decimal to binary encoder** for BCD digits. Since it is a BCD encoder, the encoded output should have 4 bits to represent the decimal digits from 0-9.

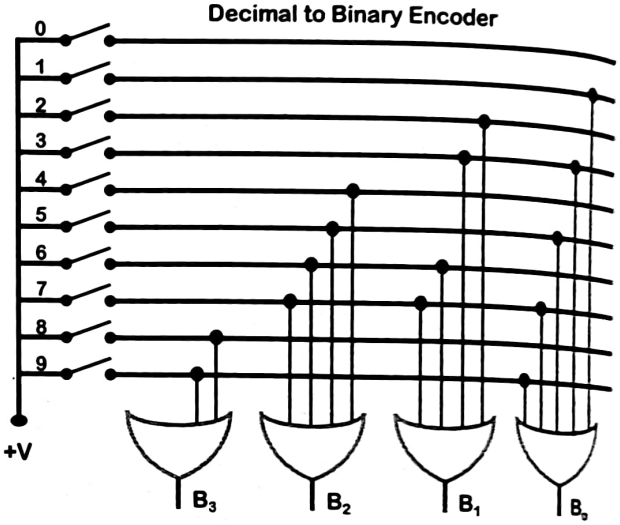
The circuit is **similar to the push button switches** in a calculator. When a user presses a particular key, the key generates a BCD number corresponding to the decimal digit printed on the button.

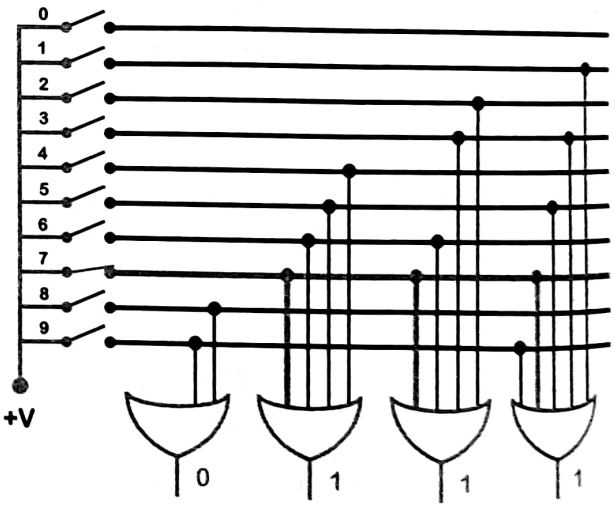
An Encoder works opposite to a Decoder and finds the binary value for a given number in another system

In the circuit shown on the right, when any one of the push-buttons is pressed, the corresponding output line gets high with **+V** volts. The **inputs to the OR gates** attached to that particular line also get high. These high inputs are transferred to outputs B_3, B_2, B_1 or B_0 depending on which OR gate/gates are high.

For example, suppose the button '**7**' is pressed. The corresponding **BCD number is '0111'**. Accordingly in the input line for '7', connection is established between the line 7 and the OR gates B_2, B_1 and B_0. As a result the corresponding outputs of the OR gates become '1', producing 0111 in the output.

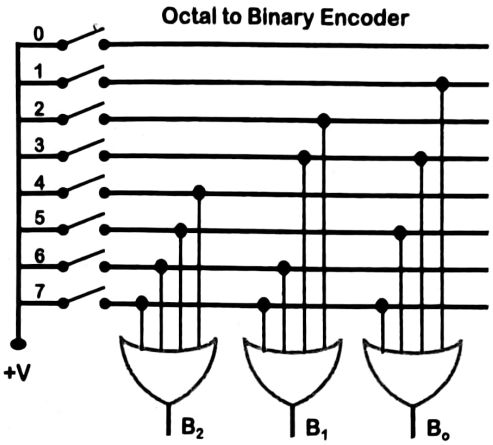
Decimal to Binary Encoder





Octal to Binary Encoder:

The circuit diagram of an **Octal to Binary encoder** is shown on the right. There are 8 input lines from 0 to 7 for the eight octal digits and 3 output lines B_2 B_1 B_0, corresponding to the three digit binary equivalent of an octal digit.



Octal to Binary Encoder

Difference between a Decoder and a Encoder

Decoder	Encoder
A decoder is used to decode a binary data to some other base like decimal etc.	An encoder is used to encode data in some other base like decimal, octal, etc. to binary
A (m to n) decoder can decode m binary input signals to a maximum $n=2^m$ decoded output signals	A (m to n) encoder can encode m input signals in a given base to a maximum of $n=\log_2 m$ binary digits

Difference b/w Decoder and Encoder

The Fact File

- When the output of a digital circuit depends on the present state of inputs only and not on the previous set of inputs, then that circuit is known as a combinational circuit
- **The basic constituents of a combinational circuit are a set of input variables, logic gates to implement the logic, and a set of output variables**
- A Half Adder circuit is a logical circuit that is used to add two binary digits. It adds two binary digits X, Y and produces a Sum and a Carry-Out output
- **The sum and carry expression for a Half Adder is given by: Sum = $\bar{X}Y + X\bar{Y}$ and Carry = XY**
- The sum portion of a Half Adder is basically an XOR operation. Thus we can replace the Sum circuit by an XOR gate also. Therefore sum = X⊕Y for a Half Adder
- **A Full Adder circuit is used to add 3 binary digits. These include the X, Y, and Carry-In or C_{in} bits, and produces a Sum and a Carry-Out as output**
- Full Adder Sum = $\bar{X}Y\overline{C_{in}} + X\bar{Y}\overline{C_{in}} + \bar{X}\bar{Y}C_{in} + XYC_{in} = X \oplus Y \oplus C_{in}$ & Carry = $XY+XC_{in}+YC_{in}$
- **Alternative expression for Carry = $(X \oplus Y)C_{in} + XY$ in a Full Adder circuit**
- A Full Adder circuit can also be designed using Half Adder circuits. To do this we require two Half Adder circuits and an OR gate
- **A Half Subtractor circuit is used to subtract two binary digits. In carrying out subtraction we have the difference D and the borrow B as outputs**
- For a Half Subtractor circuit, the Difference $D = \bar{X}Y + X\bar{Y}$ and the Borrow $B = \bar{X}Y$
- **A Full Subtractor has 3 inputs viz. X (minuend), Y (subtrahend) and the borrow-in Bin from the previous stage of subtraction. At the output we have the difference D, and a borrow bit Bo generated by the current subtraction**
- For a Full Subtractor, the Difference = $\bar{X}Y\overline{B_{in}} + X\bar{Y}\overline{B_{in}} + \bar{X}\bar{Y}B_{in} + XYB_{in}$ & Borrow = $\bar{X}B_{in} + \bar{X}Y + YB_{in}$
- **Alternative expression for Borrow = $(\overline{X \oplus Y})B_{in} + \bar{X}Y$ in a Full Subtractor circuit**
- A multiplexer means many-into-one, i.e. it is a circuit with many inputs but only one output. If there are n inputs and m control signals to select those inputs then depending upon a particular combination of the m control signals, only one of the n input signals is selected and transmitted to the output at a time
- **The Multiplexer output can be given as $F = \overline{C_1}\,\overline{C_0}\,I_0 + \overline{C_1}\,C_0\,I_1 + C_1\,\overline{C_0}\,I_2 + C_1\,C_0\,I_3$**
- A Multiplexer can be used to create a Common Bus System or to realise an SOP function
- **A De-multiplexer means one-into-many. It is a circuit which has one input and many outputs. In a 1:n DEMUX there is a single input and 'n' outputs, with 'm' controls signals used to select a particular output line**

- For a 1x4 DEMUX the outputs are given by $D_0 = \overline{C_1}\ \overline{C_0}\ I$, $D_1 = \overline{C_1}\ C_0\ I$, $D_2 = C_1\ \overline{C_0}\ I$, $D_3 = C_1\ C_0\ I$
- **A Decoder is similar to a Demultiplexer, with the exception that there are no data input lines. Like a DEMUX, 'm' control signals can be used to activate 2^m outputs in a Decoder**
- For a 2x4 DECODER the outputs are given by $D_0 = \overline{I_1}\ \overline{I_0}$, $D_1 = \overline{I_1}\ I_0$, $D_2 = I_1\ \overline{I_0}$, $D_3 = I_1\ I_0$
- **A Decoder can be used to decode a binary signal to a different number system or to activate certain circuits based on the decoded input signal. It is used as a Binary to Decimal Decoder, Address Decoder, Instruction Decoder etc.**
- An encoder is used to convert or encode an active input signal to a binary coded output signal. Thus functionally it is opposite to that of a Decoder
- **A binary to decimal decoder converts a binary signal to its corresponding decimal number**
- A binary to octal decoder converts a binary signal to its corresponding octal number

Review Questions

Q1. **Fill in the blanks:** **1 each**

a) To add two binary digits the logical circuit that is used is known as a HA circuit.
b) For a Half Adder, the Sum of the X & Y bits = $X \oplus Y$ and carry C = XY.
c) To accomplish the function of adding 3 binary digits, a F.A circuit is used.
d) The sum expression of a Full Adder can also be written as $X \oplus Y$.
e) The difference expression for a Full Subtractor is given by D = $X \oplus Y$.
f) A multiplexer means many into 1.
g) A multiplexer circuit has 1 output(s).
h) A De-multiplexer means 1 to many.
i) A De-multiplexer circuit has 1 input(s).
j) A Decoder can convert data from binary to decimal.

Q2. **Multiple Choice Questions. Select any one from the four options.** **1 each**

i) When the output of a digital circuit depends on the present state of inputs only and not on the previous set of inputs, then such a circuit is known as a:
a. combinatory circuit b. combinational circuit c. combiform circuit d. combidigital circuit

ii) The basic operation of a half adder circuit is to add:
a. 3 binary digits b. 2 binary digits c. 4 binary digits d. None of these

iii) The sum expression for a half adder which adds the digits A and B is given by:
a. AB b. $\overline{A}\,\overline{B} + A\,B$ c. $A\overline{B} + \overline{A}B$ d. A+B

iv) For a half adder circuit adding the bits A and B, the carryout bit expression is given by:
a. $\overline{A}\,B$ b. $A\,B$ c. $A\,\overline{B}$ d. $\overline{A}\,\overline{B}$

v) To add 3 binary digits the combinational circuit used is called a:
a. Binary Adder b. Full Adder c. Whole Adder d. 3 Digit Adder

vi) The sum output of a circuit adding the binary bits X, Y, and carry-in C_{in} is given by:
a. $(X \oplus Y)\,C_{in}$ b. $X \oplus Y + C_{in}$ c. $X \oplus Y \oplus C_{in}$ d. $(X \oplus Y)\,C_{in} + X\,Y$

vii) The carry-out output of a circuit that adds 3 binary digits X, Y, and C_{in} is given by:
a. $(X + Y)C_{in} + X\,Y$ b. $X \oplus Y \oplus C_{in}$ c. $X \oplus Y + XYC_{in}$ d. $(X \oplus Y)C_{in} + X\,Y$

viii) How many half adder circuits are required to design a full adder circuit?
a. 1 b. 2 c. 3 d. 4

ix) A half subtractor circuit can subtract a maximum of how many binary digits?
a. 2 b. 3 c. 4 d. 5

x) The difference output for a half subtractor circuit subtracting the bit B from bit A is given by:
a. A+B b. $\overline{A}\,B$ c. A-B d. $A \oplus B$

xi) The borrow output for a half subtractor circuit subtracting the bit B from bit A is given by:
a. $A \oplus B$ b. $A\,B$ c. $A\,\overline{B}$ d. $\overline{A}\,B$

xii) How many inputs are there for a full subtractor circuit?
a. 1 b. 2 c. 3 d. 4

xiii) The difference of a full subtractor circuit is given by an expression like:
a. $(X \oplus Y)\ B_{In}$ b. $(X \oplus Y)\ B_{In} + X\,Y$ c. $X \oplus Y + B_{In}$ d. $X \oplus Y \oplus B_{In}$

xiv) The borrow-out bit for a full subtractor circuit can be given by an expression like:
a. $(\overline{X \oplus Y})\ B_{In} + \overline{X}\,Y$ b. $(\overline{X + Y})\ B_{In} + \overline{X}\,Y$ c. $(\overline{X \oplus Y}) + \overline{X}\,Y\,B_{In}$ d. $X \oplus Y + B_{In}$

xv) A full subtractor can be formed by using two half subtractors and a:
a. NAND gate b. AND gate c. NOR gate d. OR gate

xvi) A multiplexer circuit has:
a. one-input, many-outputs b. one-input, one-output
c. many-inputs, many-outputs d. many-inputs, one-output

xvii) In a multiplexer, m control bits can be used to select a maximum of how many input lines?
a. 2^m b. 2m c. m^2 d. m

xviii) The output of a 4x1 multiplexer circuit with inputs I_0, I_1, I_2, I_3 can be given as:
a. $I_0 + I_1 + I_2 + I_3$ b. $\overline{C_1}\,\overline{C_0}\,I_0 + \overline{C_1}\,C_0\,I_1 + C_1\,\overline{C_0}\,I_2 + C_1\,C_0\,I_3$
c. $C_1C_0\,I_0 + C_1C_0\,I_1 + C_1C_0\,I_2 + C_1C_0\,I_3$ d. $(\overline{C_1}\,\overline{C_0} + \overline{C_1}C_0 + C_1\overline{C_0} + C_1C_0)(I_0 + I_1 + I_2 + I_3)$

xix) A de-multiplexer circuit has:
a. many-inputs, one-output b. one-input, many-outputs
c. one-input, one-output d. many-inputs, many-outputs

xx) In a de-multiplexer m control bits can be used to select a maximum of how many output lines?
a. m^2 b. m c. 2m d. 2^m

xxi) A decoder circuit has:
a. one-input, one-output b. one-input, many-outputs
c. many-inputs, many-outputs d. many-inputs, one-output

xxii) A decoder circuit can be used to decode:
a. octal data b. binary data c. decimal data d. hexadecimal data

xxiii) An encoder circuit is functionally opposite to a:
a. de-multiplexer b. decoder c. multiplexer d. none of these

xxiv) A decimal to binary encoder circuit has how many input lines?
a. 2 b. 8 c. 10 d. 16

xxv) A decimal to binary encoder circuit has how many output lines?
a. 1 b. 2 c. 3 d. 4

Q3. Short Answer type questions: **1 each**

SA

i) Maximum how many bits can be added using a half adder circuit? 2
ii) Maximum how many bits can be added using a full adder circuit? 3
iii) Write the sum and the carry outputs of a half adder adding the bits A and B.
iv) Write the difference and the borrow outputs of a half subtractor subtracting bit B from A.
v) State one difference between a half adder and a full adder circuit. bits count / SOP exp
vi) State one difference between a half subtractor and a full subtractor circuit. bits count / SOP exp
vii) How many half adders are required to create a full adder circuit? 2
viii) What is the function of a multiplexer circuit?
ix) How many inputs are possible for a multiplexer with 2 control lines? 4
x) State any one use of a multiplexer circuit.
xi) What is the function of a de-multiplexer circuit?
xii) How many outputs are possible for a de-multiplexer with 2 control lines? 4
xiii) State any one use of a de-multiplexer circuit.
xiv) What is the function of a decoder circuit?
xv) State one use of a decoder circuit.
xvi) How many decoded outputs are possible for a decoder circuit with 3 input lines? $2^3 = 8$

xvii) What is the function of an encoder circuit?

xviii) How many outputs are required for a decimal to binary encoder? 4

LA

Q4. **Long Answer type questions:** **7 each**

i) Which logic gate can be used to get the sum output of adding two binary digits? Draw the truth table of a half subtractor circuit. Draw the circuit diagram of a half adder. 1+2+4

ii) State any two functions of a multiplexer circuit. Show how to form a full adder circuit using half adder circuits with the help of a diagram. 2+5

iii) Draw the truth table for a half adder circuit. Write the expression for the sum output of the adder. Draw the circuit diagram of a half adder circuit using the basic logic gates. 2+1+4

iv) Write the expression for the sum and carry outputs of a full adder circuit. How many half adders are required for a full adder? Draw the block diagram of a 4 bit adder circuit. 2+1+4

v) Draw the truth table for a half subtractor circuit. Write the expression for the difference output of the subtractor. Draw the circuit diagram of a half subtractor using basic logic gates. 2+1+4

vi) Draw the circuit diagram of a 4x1 multiplexer circuit. Write the expression for the output of a 4x1 multiplexer circuit. State one use each of a de-multiplexer and a decoder. 4+1+2

vii) Draw the circuit diagram of a 1x4 de-multiplexer circuit. Write the expression for the output of a 1x4 multiplexer circuit. State one use of an encoder. 4+2+1

viii) State two differences between a multiplexer and a de-multiplexer. Draw the circuit diagram of a decimal to binary encoder. How can you convert a de-multiplexer to a decoder? 2+4+1

ix) State two differences between a multiplexer and a decoder. Draw the circuit diagram of an octal to binary encoder. Write the output expressions for a 2-to-4 decoder with inputs X, Y. 2+4+1

x) Draw the circuit diagram of a 3-to-8 decoder. State any two differences between a decoder and a multiplexer circuit. 5+2

CHAPTER 6
Operating System

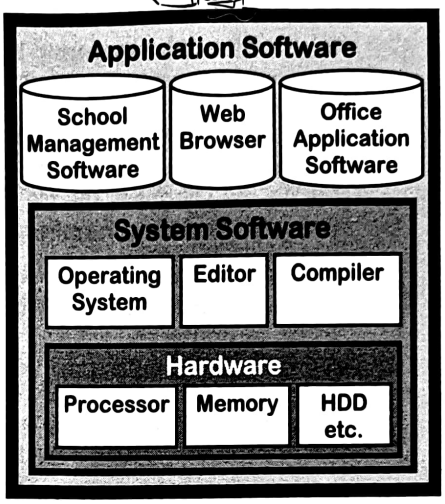
- Software and its Types 6-1
- Operating System 6-4
- The Disk Operating System (DOS) 6-11
- Windows Operating System 6-22
- UNIX Operating System 6-34
- The Linux Operating System 6-46

6.1 Software and its Types

Hardware form the building block of a computer. It consists mainly of the CPU, memory module, input/output devices and storage devices. **However computer hardware cannot do anything on its own** and it has to be instructed to do any desired job. A sequence of instructions needs to be given to the computer in a language that the computer understands to make the hardware work. Such a **set of instructions is known as software.**

Depending upon the nature of the work computer software can be divided into two main categories. These are **System Software** and **Application Software.**





System Software

- System Software

System software include software that are used to **run the computer and manage its different resources.** These help in the proper and easy use of the computer system and its maintenance. In general, system software is **much closer to the actual hardware** than the application programs

1. **Operating System:** The Operating System is the **most significant system software,** which acts as a **link between the hardware and the user.** Various application programs are written to work with the Operating System. The user interacts with application programs, which in turn interact with the computer hardware through the Operating System (OS). Examples of various OS include DOS, UNIX, Linux, Windows, Mac-OS etc.

The Operating System acts as a link between the hardware and the user

The Operating System consists of a set of programs that are used to do its various functions like Process Management, Memory Management, Input/Output Control, File Management, Security Control, Command Interpretation etc. These functions are explained below.

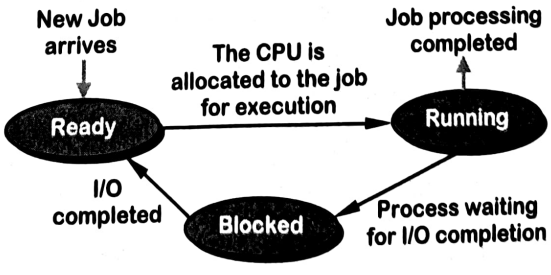
The earliest mainframe computers lacked any form of operating systems.

a) **Process Management:** When a program is executed, it is broken down into smaller units called processes or jobs. **Each process acts as a unit of work for the processor** and shares the computer resources like CPU, memory, files and input/output devices. A multiprocessing system has many processes which run together. The Process Management Module of the OS is responsible for creation, deletion and running of several processes simultaneously.

In a multiprocessing system, all processes residing in the main memory can be in one of the following three states as shown by the **3-state model** or the **Process State Diagram.**

i. **Ready Process:** Any new process entering a system must initially go to the *Ready* state. There it waits in a queue with the other processes to be processed by the CPU. Thus there can be several processes in this state.

ii. **Running Process:** Processes can enter the *Running* state via the *Ready* state only. In the *Running* state the process has control over the CPU and uses it to get executed. Though



Three state model of a process

Process State Diagram

many processes can be in the *Ready* or *Blocked* states, but **only one process can be in the *Running* state** at any instant for a single processor.

iii. **Blocked Process**: Processes normally leave the system from the *Running* state after its execution. In the event of an input/output operation, or an interrupt, the *Running* state of a process is changed to *Blocked* state. Once the input/output operation is completed, the process can again move out of the *Blocked* state to the *Ready* state for execution.



Memory Management

b) **Memory Management**: To optimise CPU utilisation a computer system loads several programs into its main memory. Every program in turn is broken down into a number of processes before execution. When a **new process is created, a portion of the main memory is allocated to that process**. Similarly when a process execution is over, it is removed from the main memory and the space reallocated to some other process.

As the total available memory space (i.e. RAM) is constant, the operating system uses various memory management schemes, like virtual memory, to meet the simultaneous memory requirement of several processes. The Memory Management Module of the OS is **responsible for the allocation and de-allocation of memory space to various programs**.



Device Management

c) **Device Management**: Several input/output devices (like keyboard, mouse, monitor, printer, etc.) are attached to a computer system. **Each device has its special hardware and specific software** called **the device driver** which can operate that particular device. Any other program wanting to use that particular device uses the device driver program to access the device. The device driver in turn communicates with the input/output hardware.

The **Device Management Module keeps track of the input/output requests** from various processes, issues commands to the input/output devices and ensures correct data flow between them.



File Management

d) **File Management**: A computer **stores data in units called files**. When a user creates a file, the OS gives a unique file name to the file and allocates space for storing the file in the system. To read the file the OS opens the file and retrieves the data stored in the file. The File Management Module deals with **creating, naming, storing, retrieving, organising and security of files**.

Files can be grouped together and stored in units called folders. A particular folder can again have sub-folders, forming a tree structure. The OS is responsible for creating and maintaining the folder structures and keeping track of which file is stored in which folder. When working with several files the OS also keeps track of which file is opened for reading and which one for writing.



Security Management

e) **Security Management**: Unauthorised access to information stored in computers is not desired. Each operating system has a security mechanism to protect the files and programs running in the computer from unauthorised access. The Security Management Module of the computer is responsible for **protecting the computer system from misuse and ensures a smooth run**.

File security can be provided by using individual or group level **passwords**. A user or a group of users with only the correct password can access a file. Moreover a file can also have a **read-only or read-write access**. Accordingly a user with a read only access will not be allowed to write into a file.

f) **Command Interpretation**: A user interacts with the operating system by means of several commands provided by the OS. The Command Interpretation Module takes care of **interpreting the user commands and making the system resources to handle the requests**.



Text Editor

2. **Text Editor**: A text editing software is used to type in some text and store it. One can open the text file in the editor for editing purpose also. This utility is used mostly by program developers who use it to type the source code of a high level language. The source code written using the editor can then be translated to form the executable program.

The first generation of editors were called **line editors** and could be used to edit a full line of text. Even to modify a single character the entire line had to be re-entered. The MS DOS text editor **EDLIN** is an example. Next came **screen editors**, were the cursor keys could be used to move up or down the screen and edit the file. UNIX operating system's **vi editor** is an example. At present we have **window based text editors**. A mouse can be used with such an editor to easily navigate through the file and edit it.



Translators

3. **Program Translator**: These include programs like assemblers, compilers, and interpreters. The main purpose of these programs is to translate a program written in assembly language or in a high level

language (like BASIC, C++ etc.) to machine language that the computer understands. The different types of translators used for this purpose are discussed below:

a) **Assembler**: An Assembler is used to translate a program written in **Assembly Language to Machine Code**. Since an assembly program depends on the machine architecture, thus assemblers are also machine dependent. The input to the assembler is an assembly language program or a **source program**, and the output is a machine language program also called an **object program**. There is a **one-to-one correspondence** between an assembly language statement and its machine code generated. **Netwide Assembler** or **NASM** is an example of an assembler.

Assembler

Given an assembly language program, an assembler first checks if the program is **syntactically correct** or not. Next it checks if all the **identifiers and labels used in the program are properly defined**. If everything is ok, only then is the program converted to its proper machine code.

It may be required to **scan the source code twice** by the assembler to produce the final object file. Accordingly we have a **two-pass assembler**. In a two pass assembler, the first pass of the source code is used to process the various symbols used in the program and the second pass is used to generate the actual machine code.

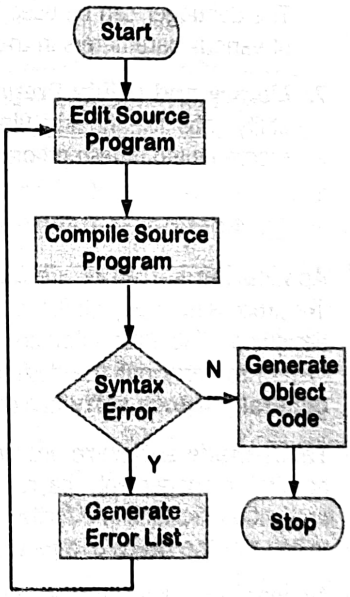
b) **Interpreter**: An Interpreter is used to **translate a program** written in a high level language to machine code **and simultaneously execute the converted code**.

Interpreter

In an interpreter, each statement from the high level language is **translated and executed one statement at a time**. In this way the whole program gets executed. This is unlike a compiler where the entire source code is translated and converted into an object program in one go. Since no object code is prepared and saved in case of an interpreter, **every time a program is to be run it needs to be translated or interpreted**. Thus it takes more time to run an interpreted program as compared to a compiled program. Since the program is executed by interpreting the program line by line, thus an **error can be detected more easily** and rectified. Original BASIC is an example.

c) **Compiler**: A compiler is a **language translator** that first takes as its input the source code from a high level language. The **entire program is then translated and saved** as a machine language code called the **object program**. In case the program needs to be modified, the source program needs to be reloaded, modified and compiled again. However, a compiler only creates an object program. It does not execute the program.

Compiler



Apart from compiling a source program, **compilers also check syntax errors** and **generate a list of error messages** when it finds errors and does not produce an object code unless the errors are rectified. However a compiler cannot detect logical errors.

Compilers are language specific, i.e. each high level language will have its own compiler program. Thus one cannot use a C++ compiler to compile a source code written in FORTRAN. The flowchart of the compilation process is given above.

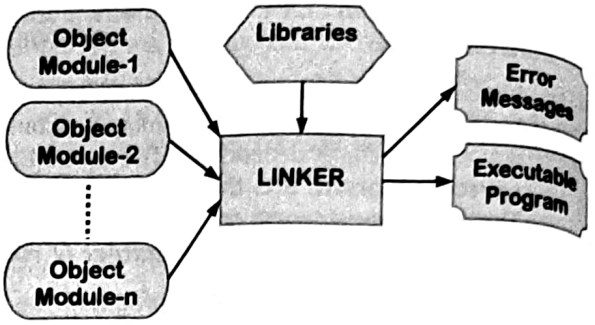
Difference between Interpreter and Compiler

Difference b/w Interpreter & Compiler

Interpreter	Compiler
1. The source code is translated a statement at a time and run	1. The whole source code is translated before it is run
2. The translated program is run automatically after it is interpreted	2. The compilation creates an object code that needs to be linked before it can be run
3. Every time the program is run, it needs to be interpreted	3. A compiled program has to be compiled only once and run as many times required
4. As each statement is translated one by one, it is easy to find errors in a translated program	4. As the whole program is compiled at once, it is difficult to locate the error
5. Running time of an interpreted program is longer	5. The running time of a compiled program is less
6. Example of interpreted programs include BASIC	6. Example of compiled program include C, C++

Linker

4. **Linker**: A large program is usually written by a team of programmers. A particular programmer can write the code for one or more modules that form a part of the entire program. Each module can then be separately compiled to produce the respective object codes or object programs. All such object programs are then combined together to produce the final program. A **Linker is system software that is used to combine or link two or more such object programs** to produce the executable form of the final program. The flowchart given on the right shows the working of linker software.



During writing a program, the programmer may include codes from the **inbuilt library of pre-compiled functions** for doing some specific jobs. The linker in that case is also responsible for **linking the object codes for those library functions** to the original program.

5. **Loader**: The executable program code produced by the linker is stored in the secondary storage like the hard disk. To run the code, the **program needs to be loaded into the primary memory for execution by the operating system**. The purpose of the loader is to do this job. In the process it takes help of another software called the relocator. The **relocator is used to adjust the relative addresses of the different sub-routines** in the executable program as per the requirement by the primary memory. The loader forms a part of the operating system nowadays.

6. **Debugger**: A debugger is a system software that is basically used to debug i.e. remove errors from the source code of a program. It can be used to run a program in a controlled manner to check for any error. The debugger can be used to make a running program halt at predefined check-points to view the values of various parameters in the program. This helps to detect and rectify an error in the code, if any.

7. **Library and Utility Programs**: These are common set of library programs that are used to do certain utility jobs like finding files, compressing files, ordering files, disk defragmenter programs, anti-virus programs etc. These programs are usually part of the utility functions provided by the Operating System.

Application software

- Application Software

Application software are **user developed programs** that are made to **carry out some specific jobs** like processing a payroll file, solving equations for scientific application, processing language, or doing graphic designing. The user interacts with these application programs, which in turn interact with the operating system to carry out the desired work. These can be **tailor-made software** serving a specific purpose or **general purpose software** developed for some general use.

Tailor-made software are **written for specific purposes** like stock management, school management, hospital management, payroll calculation etc. Programs to control the working of specific automated production facilities in a factory also fall in this category. Therefore tailor-made application programs are used to cater to the needs of a particular company, manufacturing unit or organisation.

Another type of application software are **general purpose software**. These include utility **software** that are used **to do common business oriented jobs** or **scientific analysis**. These software are used to do word processing, spread sheet processing, database handling, presentations, computer aided designs, electronic mailing, desktop publishing etc. These are usually available as bundled packages like MS Office, Corel Suite, Adobe Smart Suite, AutoCAD etc. The major advantages of such software are that they are pre-tested, feature rich, professionally developed programs that are usually easy to use from a user point of view.

Operating System

6.2 Operating System

We have seen that an operating system forms the most important system software in a computer system. In this section we will discuss about the various types of operating systems, and some of their common features.

- Categories of Operating Systems

Based on the **way a user interacts with a computer** and the way the computer carries out the jobs assigned to it, one can broadly classify operating systems into the categories as described below:

a) **Mainframe Operating Systems**: These operating systems serve mainframe computers which can consist of **thousands of GB of data**. These computers can be used as web servers, business servers, in railway booking office, stock exchange, etc. where a **large number of people access the computer at any time**. Thus operating systems for mainframes are specially developed for handling large volumes of data and can process many jobs at a given time. Example of such an operating system is **OS/390**.

The first operating system used for real work was **GM-NAA** I/O, developed in 1956 by General Motors for IBM 704.

b) **Batch Processing Operating Systems**: Such an operating system is used for **processing a set of jobs without any human intervention**. The jobs submitted by the users are collected by the computer in a batch and are put in an input queue. The computer then processes each job one after the other without any interaction from the user.

In the early days of OS, each time a manufacturer brought out a new machine, there would be a new operating system.

A batch operating system inputs a set of separate jobs and processes each job one after the other in a first-come-first-served basis. To separate one job from another, special languages called **job control languages** (JCL) containing control statements were introduced. The control statements were used by the operating system to identify new jobs and determine the resources needed by the job during execution. Usually every program or data set were preceded and followed by such language statements.

When a job is completed its output is usually printed or stored in a file. However a delay is there between the job submission time and the job completion time. This delay is called the **turnaround time**.

Batch processing is suitable for jobs requiring a large number of calculations without human interaction. Examples include payroll file processing, weather forecasting, statistical analysis etc.

However a batch operating system has the **following disadvantages**:

- A batch operating system **does not interact with the user** when a job is under processing and the users have no control over any intermediate result of the processing.
- The **turnaround time** can also be high which may not be desired at times.

Master Control Program operating system was the first OS to be written entirely in a high level language ESPOL, and for the first time had features like virtual memory.

c) **Multiprogramming Operating Systems**: Multiprogramming means **interleaved execution** of two or more **different independent jobs** or programs by the same computer. Such a type of operating system allows more than one program to be stored in the primary memory simultaneously. Proper memory management techniques are needed to do this. In case several jobs are in the ready state, then the operating system should be able to decide the order in which the jobs need to be executed. Moreover when several processes are running together, each one should run independent of the other.

- **Multitasking Operating System**: A multitasking operating system **can handle multiple tasks together by applying multiprogramming techniques**. When multiple programs are running in a computer, each is broken down into a number of smaller processes. When one process of a particular program has finished using the CPU and is ready to do I/O operations, the CPU is allocated to a process of another program which needs execution. The CPU switches from one process to another almost instantaneously and to the user gives the impression of simultaneous operation of all the programs. Examples of multitasking operating systems are UNIX, Windows, Linux etc.


Multitasking OS

- **Multiuser Operating System**: Such an operating system **allows multiple users to access a computer through more than one terminal**. Railway booking systems use such an approach, where a number of users can access the main server computer from various terminals spread across a wide area network. Examples of such operating systems are UNIX, Linux, Windows2000 etc.
- **Multiprocessing Operating System**: A hardware implementation of a CPU may have **multiple independent processors that work in parallel**. Such computer systems are required for advanced scientific research and commercial applications which require a great deal of processing power. An operating system that supports such a hardware configuration and helps to share the limited resources of the computer amongst the different processors is called a Multiprocessing Operating System. Examples include UNIX, Linux, Windows.
- **Time Sharing Operating Systems**: Such an operating system allows various users to **share the total available time of the CPU**, memory and other resources of the computer system under its supervision. Timesharing is the process by which **many users can simultaneously use a computer system** such that each user is given the impression that he is using his own computer. A computer system achieves this by using a **task-scheduling algorithm** that allocates a very short period of the CPU time (known as a **time slot**) to each user one by one. Once the last user is accessed, the process again starts from the first user. Examples of such systems are the VAX/VMS and UNIX workstations.


Time Sharing OS

Real Time OS

d) **Real Time Operating Systems (RTOS)**: These types of operating systems are **used in places where the response time required for data processing is critical**. Examples of such processes are air traffic control, nuclear power plant monitoring, flight control of artificial satellites, real time simulations etc. Such types of operating systems are designed to respond to external interrupts that require immediate response of the computer system. Examples of such OS include LYNX, RTX etc.

Network OS

e) **Network Operating Systems (NOS)**: This type of **operating system is used in computers connected in a computer network**, like a Local Area Network (LAN). Such an OS is capable of handling not only the resources of the computer system where it is installed, but also **handles communication from other computers** connected to it. To access resources from other computers, a user may have to log into the remote computer using proper password. One of the main advantages of using a network operating system is to **utilise common hardware resources** like printers connected to the network.

Examples of network OS include Windows2008 Server, LINUX etc.

f) **Distributed Operating Systems**: In such a system, several computers that are connected together **process a given job by sharing the job load** between them. An operating system that supports distributed processing should have special features to coordinate the operations and flow of data between the participating computers.

Unlike a network OS, in a distributed system, **the user remains unaware that the job is being processed in a distributed manner**. To the user it appears that the processing is being done in a single processor system. The operating system carries out all the operations required to seamlessly perform the processing in a distributed manner. AMOEBA is an example of such an operating system.

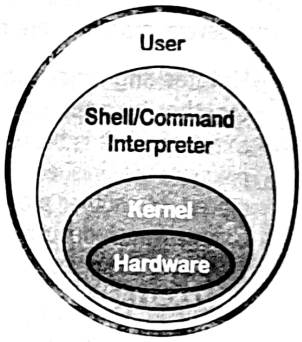
g) **Personal Computer Operating System**: These types of operating systems are used in PCs and are usually meant for home use. Such an OS does not have the full features of a normal operating system and have **features more suited for home use**. Examples include Windows 2007 Home edition etc.

- General structure of an Operating System (OS)

Modern operating systems have a layered structure. The bottom-most layer forms the hardware interface part of the computer and the outermost layer forms the user interface. In between these two layers are the other layers of the operating system. Each layer is responsible for a set of functions and communicates with the layers below and above it.

OS Kernal

a) **The Kernel** is the innermost layer and is the **central controlling part of the operating system**. It always resides in the main memory and directly communicates with the hardware of the computer. A typical kernel contains programs that are used for the **basic functions of an operating system** like process management, memory management, input/output device management. It also includes low level security features and **interprets the commands** supplied to it by the command line or GUI of the OS. The kernel is also that part of an operating system that a user cannot replace or modify.



OS Shell

b) **The Shell** is the layer next to the Kernel. It usually **serves as the user interface** through which the users interact with the operating system and **acts as a command interpreter**. The shell also **manages running of multiple application programs**. The shell can be a command line interface (like in DOS) or a graphical user interface (like in Windows or Mac). A given operating system can have both a command line and a GUI shell, or more than one command line shells as in UNIX. The user can switch between the two as per his convenience.

In some operating systems like UNIX, the shell also provides the platform to write programs that can be used as user defined commands. Several such commands can be combined to form a **shell script** which can be used to perform a specific operation.

Types of OS Interfaces

- Types of User Interfaces in an Operating System

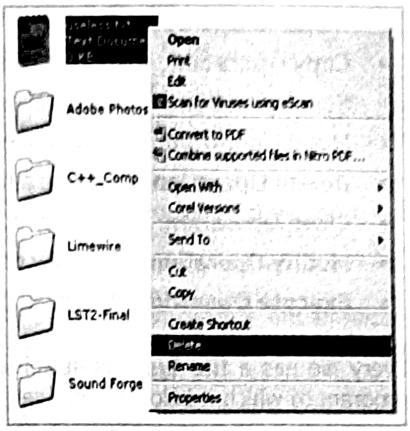
An operating system provides an interface through which the user interacts with the system. Common types of interfaces include **Command-Line Interface** (CLI) or **Character User Interface** (CUI) like DOS, and **Graphical User Interface** (GUI) like Windows and Mac OS. In a CUI, the commands are to be typed to be executed. In a GUI a pointing device can be used to click on icons or menus to execute a command.

a) **Command Line Interface**: Also called a Character User Interface, here the user interacts with the operating system by **typing commands** on a command line. Various commands need to be typed for carrying out various jobs like creating, deleting, editing, renaming or printing a file. Usually **each command represents an executable program**, which is run when the command is typed with the proper parameters. This program in turn carries out the required task based on the parameters supplied. For example, when the DOS command '*del useless.txt*' is typed and run, it runs the executable command file **del** to delete the text file *useless.txt* from the current folder.



Command Line Interface

b) **Graphical User Interface**: Alternatively a user-friendly graphical interface can be used to carry out the required interfacing. A GUI provides a **window and menu based graphical interface** that can be operated by using a pointing device like a mouse. The mouse can be used to point and click on command button pictures or icons to execute commands, open folders, or run programs. Thus users no longer have to remember commands to type and execute them.



Graphical User Interface

Moreover, as **similar type of icons or pictures are used to represent the same type of command**, the user does not have to remember various command names for doing the same job in different programming environments.

Thus to delete the file *useless.txt*, the user simply may have to click the right mouse button over the file icon and select the *Delete* option from the **drop-down menu displayed**. By clicking on that option, the file will get deleted.

The **XEROX Alto** was an early personal computer developed in 1973 and was the first computer to use a mouse driven Graphical User Interface.

Differences between CUI and GUI

Character User Interface	Graphical User Interface
1. Commands have to be typed in the command line to execute them	1. One can point and click on icons to execute commands
2. Commands need to be remembered to be able to type and execute them	2. Commands need not be remembered as they can be executed by simply clicking on icons
3. Different programs may have different commands for the same job. While Ctrl+X is used in the EDIT application of DOS, the Shift+Del key is used in the Turbo-C program interface to CUT a portion of text	3. Usually an universal icon is used to represent a particular command. For example to CUT a portion of text the symbol ✂ is used for an icon in all GUI based applications
4. Drag and drop features are usually absent. This makes the execution of certain commands lengthy	4. Drag and drop features are available which make certain command execution easier
5. These are less user friendly	5. These are more user friendly
6. Example: DOS	6. Example: Windows, Mac

Difference b/w CUI and GUI

- File and Directory System

The operating system maintains a directory or folder structure to store data in a computer storage media like a hard disk or a CD. The data is stored by the computer system in the form of files.

A **file is a collection of related data or information** that is stored in a secondary storage media. A file can be either a **data file** or a **program file**. A data file basically stores data or information and can be numeric, alphanumeric, or binary in nature. On the other hand a **program file** contains code that can be executed or run. The various types of data files are discussed later.

A **file** can be characterised by the following **properties**:

- **File Name**: Each file is identified by a **file name**. The file can be accessed by using this name. Different operating systems may have different rules for naming a file.
- **File Extension**: An **optional extension name** which can be used to specify the type of the file. For example a word processed document may have an extension of doc or docx.
- **File Date and Time**: Along with the data, each file also stores information related to the **date and time of its creation** or modification.
- **File Length**: The total **byte content** of the file called the file length is also stored.

File and Directory System

- **File Protection Attributes**: A file may also have certain **file protection attributes** like read-only, archive, system or hidden, which determine the type of file access given to the user. A read-only file can only be read and cannot be modified by the user. Similarly a hidden file will not be visible to user. Usually system files are kept as hidden files so that these are not deleted accidentally.

The **various operations possible** on a file include:

- **Read Operation:** To read the information which is stored in the files.
- **Write Operation:** To insert some new data into a file.
- **Rename Operation**: To change the name of a file.
- **Copy Operation**: To make a copy of a file from one location to another, keeping the original.
- **Sort Operation**: To arrange the contents of a file in a particular order.
- **Move Operation**: To move a file from one location to another.
- **Delete Operation**: To delete a file.
- **Close Operation**: To close an opened file.
- **Modify Operation**: To modify or change the contents of a file.
- **Execute Operation**: To run an executable file.

Every file has a **file type as specified by the extension of the file**. It basically states the application program to which it belongs. The file extension is used to tell the computer the application program that can be used to retrieve the contents of the file. For example a file with some graphic content like a drawing or painting or a photograph can be opened by the Paint or the Photoshop application program.

Data File Types

Therefore data files are classified according to the way application programs use them. A file management system in general supports the **following categories of data files**:

a) **Transaction File**: Such a file is used to store input data temporarily until it is processed. For example, a transaction file may temporarily store the weekly data of employees in an organisation. Such information is later required while processing the payroll file.

b) **Master File**: Such a file contains all the current data relevant to an application. Master files contain descriptive data which are permanent in nature, such as ID, name and address of an employee in an organisation. The data in transaction files is used to update the master files. For example, to calculate the weekly pay of an employee, the weekly data from the transaction file is combined with data from master file by the payroll program. The master file is finally updated with the new data from the transaction file.

c) **Output File**: Some applications use multiple programs for data processing. In such an application, the output produced by one program is used as an input for the next program. Here the temporary output of the first file is stored in a file called the output file. The second program in turn inputs from this file.

d) **Report File**: Data processing applications need to generate various types of reports based on user queries. A report file contains a soft copy of such a report. The report file is usually stored in an electronic format before it is printed to get a paper copy.

e) **Backup File**: Such a file keeps a copy of a given file as backup. In case the original file gets corrupted or deleted accidentally, then the last saved data of the file can be retrieved from the backup file.

Common File Extensions

Some common file extensions and their meaning

File Type	Description
.bat	A text file containing a sequence of operating-system commands for execution sequentially
.bmp, .gif .jpeg, .jpg	All these are extensions for graphic files like drawings, paintings, or photographs. These can be opened by applications like Paint, Photoshop, Picasa, etc.
.cab	These are cabinet files which are used to organise and store compressed installation files
.dbf, .mdb	Database file
.doc, .docx	Microsoft Word file
.exe	Executable file I.e. a program file that can be run. Can be an application program
.htm, .html	A web file written using the hypertext markup language or HTML
.inf	Typically, each .cab file contains an associated information file, with further installation information
.mp3, .mp4	Compressed audio/video file

.pdf	Portable document file
.ppt, .pptx	Microsoft PowerPoint file
.psd	A Photoshop picture file
.txt	A simple text file written using ASCII characters. Example a Notepad file
.wav	Microsoft Windows audio file
.xls, .xlsx	Microsoft spreadsheet file

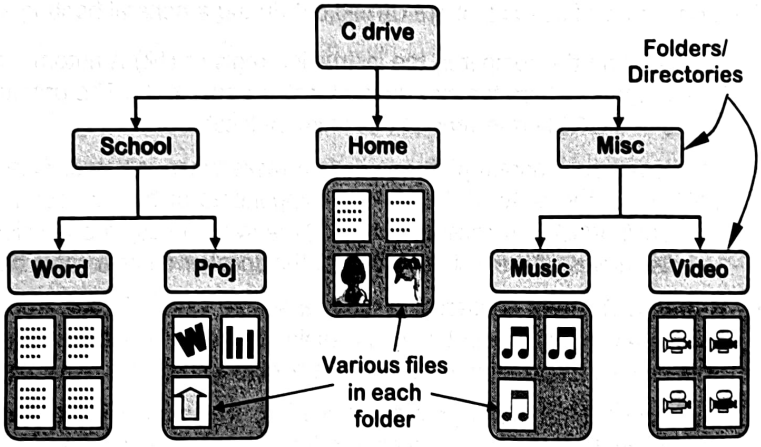
Another important aspect of a file is how the file is organised. **File organisation basically deals with how the data records are physically organised in a file.** The organisation is based on the ease of data retrieval and the nature of storage. The four commonly used **methods of file organisation are:**

Data File organizations

1. **Serial file organisation:** In a serial file, the records are stored in no particular logical sequence and are stored serially one after the other. Normally the records are stored in the file in chronological order i.e. as and when they are created.
2. **Sequential file organisation:** In a sequential file, the records are stored in a particular order with respect to a given field of the record. This field can be a key field like the primary key of the file, or a non-key field of the file.
3. **Indexed file sequential organisation:** In an indexed sequential file the records are physically ordered with respect to a search key in the file. In addition to that, a primary index for the file is also maintained. The index helps in search operations.
4. **Direct or random file organisation:** In a direct file the records in the file can be accessed directly using a special process on the search key of the file. The process helps to find the location of the record in the file at a very fast rate.

Directory structure

Directories (or folders) are used **to group other directories or files and put them under a common heading.** Each directory can have a number of entries, one per file. When a file is created, a new entry for the file is created in the directory. When the file is deleted, the entry for the file is also deleted from the directory listing. Using directories it is possible to separate and **store files under various logical headings,** thereby helping in file search.



Tree structure of Folders or Directories

All modern operating systems use a **hierarchical or inverted tree like directory structure.** In such a structure there is a main **root directory** which in turn can have several subdirectories, and so on. Each directory can also store one or more files.

The diagram shows one such directory structure. Here the **C** drive forms the root directory, which has the subdirectories '*School*', '*Home*', and '*Misc*'. While the '*Home*' sub-directory contains several files, the '*School*' subdirectory again contains two sub-sub directories namely '*Word*' and '*Proj*'. Each of these sub-sub directories in turn contains several files.

Difference between File and Directory (or Folder)

Difference b/w File & Folder

File	Folder or Directory
1. A file is a collection of data arranged in a given order	1. It is a place to store a set of related files or other folders
2. Files can have file extensions to indicate the type of the file or the application program to be used	2. Directories do not have such extensions
3. Various file organisations include – serial, sequential, indexed sequential, and direct file organisation	3. Various directory organisations include – single directory, one directory per user, and multiple directory per user organisation
4. A given file cannot contain other files	4. A given folder can contain other folders

File Allocation Table

• File Allocation Table

Data in a computer is stored in a secondary storage device like a hard disk. To store data each hard disk surface is **divided into small logical blocks**. Each block forms the **minimum storage unit for data** during file read/write operations. During a read operation, the whole block containing the required data is read. Similarly **during a write operation, the whole block is written** into the disk.

In some operating systems a single sector forms a block, while in others **several sectors form a cluster** which in turn forms a block. When a new file is created, an operating system allocates space in the hard disk to store that file in several blocks depending upon the file size.

Computers maintain on the disk, a table called the File Allocation Table or FAT indicating the block location of a file.

Computers maintain on the disk, a **table indicating the block location of a file**. This table is known as the **File Allocation Table** or FAT and helps the computer to locate a file easily by serving as an index. When a file is written to a disk, the operating system checks the FAT for an empty area, stores the file there and then writes the file location in the FAT. When an existing file is deleted, the entry of the file from the FAT is removed. The FAT therefore forms the most important area in a disk as it stores **information regarding the location of a file and the availability of free space** to store new files.

Booting

• Booting

In the early days of computing, a program had to be manually input to the computer before it could be used. The next step in computing was to automatically load the programs prepared offline, through some input device. **The procedure of starting a computer by loading the basic components of the operating system into its main memory is known as booting**. The word has come from the term "bootstrapping" which basically means pulling the computer up by its own efforts.

The **sequence of steps** that are carried out during a general booting are given below:

- After starting the computer, the instruction register (IR) is automatically loaded with a predefined memory location, from where the execution of instructions starts. The primitive bootstrap program which is usually stored in the ROM is located at this memory location.
- The bootstrap program performs certain tasks when initiated. One of its tasks is to run a diagnostic test that checks the state of the various components of the computer system. This is called the **Power On Self Test** (**POST**). It consists of testing the system bus, internal clock, video display card, RAM, keyboard, and disk drives. If the test is successful the speaker emits a short beep.
- If the POST check is passed, it reads a simple block of code from a section of the hard disk called the **boot sector** and loads it into the main memory. This block of code stores information regarding the location of the remaining portion of the bootstrap program, which is loaded next.
- Once the full bootstrap program is loaded, it can then locate the operating system kernel and device drivers and load them into memory to finally make the computer running.

Warm Booting skips the RAM test.

Depending upon the way a booting takes place, there are **two different types of booting processes** as discussed below:

Cold and Warm Booting

a. **Cold Booting**: The first process to carry out when a computer is turned on is to initialise the microprocessor. Next the **Power On Self Test** (POST) is carried out. It consists of testing the internal BUS, Internal Clock, Video Display Card, RAM, Keyboard, and Disk Drives. If the test is successful the speaker emits a short beep. Then the components of the operating system are loaded into the main memory. This is known as **Cold Booting**.

b. **Warm Booting**: If for any reason the computer hangs while working, it needs to be restarted to make it functional. A computer can be restarted by pressing the Ctrl+Alt+Del keys simultaneously. When this is done, the computer restarts by skipping the RAM test. This is known as **Warm Booting**.

Spooling

• Spooling

Spooling or **S**imultaneous **P**eripheral **O**perations **On**-**L**ine is a technique used to solve the problem of **speed mismatch between the processor and peripheral devices** like printers, keyboards etc.

The CPU works at a much faster speed than input/output devices like keyboards or printers. Thus when a slow I/O device is inputting or outputting bytes to or from the main memory, the **CPU remains idle for a large percentage of the input/output time**. Until the input/output operation is over, the CPU cannot process the next data.

Spooling reduces this CPU idle time by first placing all the data that comes from a slow input device, on the hard disk before it is loaded into the main memory. Similarly, after execution by the CPU, data in the main memory is first written to a magnetic disk before it is output through a slow output device. As interaction of the CPU with the high speed storage medium like a magnetic disk is much faster than that with the much slower I/O devices, this reduces the CPU idle time considerably.

When the CPU makes a request to use a slow I/O device like the printer, then instead of using the printer to print the required job as and when generated by the CPU, the operating system opens a special file called a spool file in a special directory called the spool directory. The **output data to be printed is then stored in the spool file**. The file is closed after the CPU finishes generating the data for printing. A special process then releases the entire data stored in the spool file from the disk to the printer in line with the slower speed of the printer.

Virtual Memory


Virtual Memory

To run a program it has to be loaded into the main memory. However a process can be loaded into the memory only when sufficient main memory is free to load the entire process. Due to this, a process may be kept waiting before starting execution till sufficient main memory is free. This delays the execution of the process. Moreover if the main memory size is smaller than the process size then the process can never be loaded into the main memory and executed.

Virtual Memory allows the execution of processes even in case of insufficient free main memory by using secondary storage.

A memory management scheme called **Virtual Memory** overcomes this problem. It allows the execution of processes even in case of insufficient free main memory. To do this, a portion of the memory of a high speed secondary storage device like a hard disk is used along with the main memory. The operating system uses this secondary memory as an **additional memory area** to the main memory. A **special process called swapping** is then used to transfer blocks of data between the secondary storage and the main memory (RAM) as per requirement. Using swapping techniques the operating system manages the two memory areas in such a manner that users feel that they have access to a single large main memory.

6.3 The Disk Operating System (DOS)


DOS

DOS or Disk Operating System was introduced jointly by Microsoft and IBM during 1981 and hence was also called **MS DOS**. It was a popular operating system used by personal computer users. **DOS is a single-user, single-tasking operating system** which means that the functions provided by the basic kernel of DOS can be used by only one program at a time. It offers a **command line character user interface** through which the user has to type in the commands to execute them.

DOS boots by loading its BIOS from the system file IO.SYS which in turn loads the DOS kernel from the system file MSDOS.SYS. The kernel next executes the CONFIG.SYS file which is used for configuring the DOS shell by opening the COMMAND.COM file. The CONFIG.SYS file also contains **commands to customise DOS** by performing specific functions and **loading** external software like **device drivers**. The shell then opens a start-up batch file called the AUTOEXEC.BAT. It is a special batch file written in plain ASCII text and contains instructions for various **start-up operations** that a user wants to **perform automatically** every time the computer starts.


Internal & External Commands

The commands given to DOS are in general computer programs. Depending upon the location of the programs, these are classified as **Internal and External commands**. Once DOS has been started up i.e. the computer has booted, certain important commands are automatically loaded into the computer's memory and are available for use at all times. These are called the Internal Commands. The **internal commands form a part of the** COMMAND.COM **program**. These command programs are not displayed at the time of listing the files. Examples of **some internal commands** are:

CLS, DATE, TIME, DIR, CD, MD, RD, COPY, DEL, RENAME etc.

External commands on the other hand are command **programs that are kept in the disk until they are needed**. When one uses an external command, DOS has to refer to the disk or path where these programs are stored to run these programs. These programs are displayed at the time of listing the files. Examples of **some external DOS commands** are:

DISKCOPY, SYS, FORMAT, XCOPY, MOVE, BACKUP, RESTORE, DELTREE, ATTRIB etc.

Whenever a DOS command is entered, the COMMAND.COM checks its list to see if the command is an internal one. If so, COMMAND.COM will itself carry out the task. If the command is not internal, COMMAND.COM will look for a program that has the same name and if found will get executed.

In the original MS DOS, a file name can be up to 8 characters long and can contain the upper and lower case alphabets (A to Z, a to z), the digits (0 to 9) and the underscore character '_' in the file name. The filename should also be a single word. In case two different words have to be used, they should be joined by the underscore character '_'. A file can also have an optional extension of up to 3 characters.

• Various Internal DOS Commands

This section discusses the various important and common internal DOS commands used.


CLS

▪ CLS

The **CLS** or clear screen command is used to **clear all the other commands** already typed in the screen and leaves the screen blank and only with the current command.

Example: C:\>CLS


DATE

▪ DATE

The **DATE** command is used to **display or modify the current date.** After typing the command, type a new date in the format dd-mm-yy or press Enter to keep the same date.

Example: C:\>DATE

Output:
The current date is: 24-May-09
Enter the new date: (dd-mm-yy)


TIME

▪ TIME

The **TIME** command is used to **display or modify the current time.** After typing the command, type a new time in the format hour-min-sec or press Enter to keep the same time.

Example: C:\>TIME

Output:
The current time is: 11:30:14.34
Enter the new time:


DIR

▪ DIR

The **DIR** or directory command is used to **list all the sub files or sub directories** under a given directory. Along with the file name, the date & time of file creation and the size is also displayed.

Example: C:\>DIR

Output:

```
Volume in drive C has no label.
Volume Serial Number is 7CB5-93F2
Directory of C:\
12-Jul-07    08:21 PM       125        AUTOEXEC.BAT
09-Jul-08    10:21 AM      <DIR>       AVPDOS
12-Jul-07    08:21 PM       138        CONFIG.SYS
16-Feb-08    08:46 PM      <DIR>       TC
14-Apr-09    10:33 PM      <DIR>       TEMP
30-Apr-94    06:22 PM     4,377,400    TURBOC.EXE
             3 File(s)    4,377,663 bytes
             3 Dir(s)     5,168,078,848 bytes free
```

Example: C:\>DIR CONFIG.SYS

Output:

```
12-Jul-07    08:21 PM       138        CONFIG.SYS
```

The above example shows how a **particular file can be searched** in a directory using the **DIR** command. The file name is to be typed along with the **DIR** command. If the file is present, its name and details will be displayed in the listing. If it is not present then the message "File not found" is displayed.

The **following switches are used with DIR** for displaying the directory structure in various ways.

/p Pauses after each screen of information.
/w Display the information in a wide list format.
/o List by files in sorted order as per the following sub-switches:

Sort-order	n	By name (alphabetic),	s	By size (smallest first)
	e	By extension (alphabetic),	d	By date & time (earliest first)
	g	Group directories first,	a	By Last Access Date (earliest first)

/a Displays files with specified attributes as per the following sub-switches:

Attributes	d	Directories	r	Read-only files
	h	Hidden files	a	Files ready for archiving
	s	System files		

/b Uses bare format (no heading information or summary).
/l Uses lowercase to display the names.

Switches with DIR

The following examples will make the use of these switches clear.

Example: C:\Temp>DIR/p

The above command displays the directory and file listing of the directory **Temp** under the **C:** drive one page at a time **(/p)**, in case the listings are too long to be displayed in a single screen.

Example: D:\Music>DIR/p/on

The above command displays the directory and file listing of the directory **Music** under the **D:** drive in both a page-wise **(/p)** and ordered by name **(/on)** manner.

Example: D:\Document>DIR/ar

The above command displays the directory and file listing of the directory **Document** under the **D:** drive which have only read-only attributes **(/ar)**.

- **MKDIR or MD**

MKDIR / MD

The **MKDIR** or **MD** command is used **to make a new directory** or folder in DOS. Along with the command, the path i.e. the location of the new directory and the name of the new directory is also to be typed as:
MD [Directory_Name]

Example: C:\>MD Temp

The above command will make a directory called **Temp** under the **C:** drive.

Example: C:\>MD D:\MyDocs\Word

The above command will make a directory called Word under the **MyDocs** directory under the **D:** drive.

Example: C:\CProg>MD School Home

The above command will make two directories called **School** and **Home** under the **CProg** directory located under the **C:** drive. Note that here the **MD command is executed from within the directory CProg.** Hence the new directories **School** and **Home** are automatically created under **CProg.** To create **multiple directories,** write the names of the new directories separated by blank spaces **after the MD** command.

C:\>MD C:\CProg\School

The above command will perform the same job of creating a folder called **School** under the **CProg** folder, but in this case is executed from directly under the **C:** drive.

RMDIR or RD

RMDIR / RD

The **RMDIR** or **RD** command is used **to remove or delete a directory** or folder in DOS. Along with the command, the path i.e. the location and name of the directory to be removed is also to be typed as:
RD [Directory_Name]

Example: C:\>RD Temp

The above command will remove the directory called **Temp** from under the **C:** drive.

Example: C:\>**RD D:\MyDocs\Word**

The above command will remove a directory called **Word** from under the **MyDocs** directory located under the **D:** drive. (Note that you cannot remove a directory in which you are currently present)



CHDIR / CD

- **CHDIR or CD**

The **CHDIR** or **CD** command is used **to change from one directory to another** in DOS. Along with the **CD** command, the path and name of the directory to move to is to be typed as: **CD [Directory_Name]**

Example: C:\>**CD Temp**

Output: C:\Temp>

The above command will move to the directory called **Temp** under the **C:** drive.

Example: C:\>**CD C:\MyDocs\Word**

Output: C:\MyDocs\Word>

The above command moves to the **Word** directory under the directory **MyDocs** under **C:** drive.

To switch to a different drive there is no need to use the **CD** command. Simply type the drive name followed by a colon ':' symbol. The following command moves to the drive **D:** as shown below:

Example: C:\MyDocs\Word>**D:**

Output: D:\>

The following command will move the current directory from **C:\MyDocs\Word** to **C:\Temp\Program**

Example: C:\MyDocs\Word>**CD C:\Temp\Program**

Output: C:\Temp\Program>

The following **CD** command can be used to **move back to the parent directory** i.e. one level up the directory hierarchy. The .. after **CD** specifies that you want to move back to the parent directory. The following example will move the current directory from **C:\MyDocs\Word** to **C:\MyDocs.**

Example: C:\MyDocs\Word>**CD..**

Output: C:\MyDocs>

The **CD** command can also be used to **move directly to the root drive from any location.** As shown below, the command here moves the current directory directly from **C:\MyDocs\Word** to **C:**

Example: C:\MyDocs\Word>**CD**

Output: C:\>



COPY

- **COPY**

The **COPY** command is used **to copy a file** from one location to a different one or make a copy of a file with a different name at the same location. The copy command is to be followed by the source file name and path and the destination file name and path as: **COPY [Source_File] [Target_File]**

The following **COPY** command can be used to copy the file **biodata.doc** located under the **C:** drive to the **Temp** directory also located under the **C:** drive.

Example: C:\> **COPY biodata.doc C:\Temp\biodata.doc**

Source file name located under C drive — Target file path and file name

Output: C:\> **COPY biodata.doc C:\Temp\biodata.doc**
1 file(s) copied.

The following **COPY** *command copies the file* **prog12.c** *located under the* **Temp** *directory in* **D:** *drive to the* ***CProgs*** *directory located under the* **C:** *drive. Note that the* **COPY** *command can be executed from any location, provided the source and target paths and file names are given.*

Example: `C:\MyDocs\Word>COPY D:\Temp\prog12.C C:\CProgs\prog12.C`

Source file path and file name — Target file path and file name

Output:
```
C:\MyDocs\Word>COPY D:\Temp\prog12.C C:\CProgs\prog12.C
1 file(s) copied.
```

The following COPY command makes a copy of the file **prog12.c** located under the **CProgs** directory under C: drive in the same place, but with a different file name (a folder can't have two files with the same name).

Example: `C:\> COPY C:\CProgs\prog12.C backup.C`

Source file path and file name — Different target file name of file copied under the same folder

Output:
```
C:\COPY C:\CProgs\prog12.C backup.C
1 file(s) copied.
```

The following COPY command simply makes a **copy of the file autoexec.bat at the same place** as the source i.e. under the C: drive but with a different file name **OldAuto.bat**.

Example: `C:\> COPY autoexec.bat OldAuto.bat`

Source file name only — Target file name only

DOS refers to **different parts** of the computer with special names called **device names**. Every port in a machine has a unique name so that software can identify and **use the ports just as file names**.

CON is the **device name** for console i.e. **Keyboard** and **Monitor**. **COM1, COM2** etc. (Communications Port #1, #2 etc.) are for **Serial Ports**. **LPT1, LPT2** (Line Printer Port #1, #2) are for **Parallel Ports**.

The **COPY** command can also be used with the above device names. The device name is treated as a file name. Thus the following command can be used to copy the characters typed from keyboard to form a file. Here the device name **CON indicates the source of the file as the input console i.e. the keyboard**.

Use of Copy Con command

Example: `C:\> COPY CON MyData.txt`

Output:
```
C:\> COPY CON MyData.txt
Sauro Ghosal
Class 11, Section E
^Z
1 file(s) copied.
```

When the above command line is typed and the enter key pressed, DOS will allow the user to type any text including new line. When typing is over, press the **Ctrl+Z** keys or F6 key to end the command. The text will be copied to the file **MyData.txt** and saved under the directory from where **COPY CON** was executed.

The same **file can also be displayed onto the screen** again by using the **COPY** and **CON** combination but now in the reversed order. When the device name **CON** is placed as the target, then **CON** refers to the monitor or VDU. Thus the following command will copy i.e. display the contents of the file **MyData.txt** on the screen i.e. the file will be copied onto the screen.

Example: `C:\> COPY MyData.txt CON`

Output:
```
C:\> COPY MyData.txt CON
Sauro Ghosal
Class 11, Section E
1 file(s) copied.
```

Similarly the following command can be used **to print the contents of the file MyData.txt** using the printer connected to the parallel port with the device name **LPT1**.

Example: `C:\>COPY MyData.txt LPT1`

Output:
```
C:\>COPY MyData.txt LPT1
1 file(s) copied.
```

- **RENAME or REN**

RENAME / REN

The **RENAME** or **REN** command is used **to rename a file or a directory**. The rename command is to be followed by the path and file/directory name to be renamed and the changed file/directory name as:
RENAME [Old_File_Or_Folder_Name] [New_File_Or_Folder_Name]

The following example will rename the file with the file name **MyData.txt** to the new name **Shouro.txt.**

```
Example:  C:\> RENAME MyData.txt  Shouro.txt
```

Old file name to change — New file name

The following example will rename the file with the file name **Prog12.C** to the new name **Sorting.C**, where the file **Prog12.C** is located under the **Temp** folder in the **D:** drive.

```
Example:  C:\> RENAME D:\Temp\Prog12.C  Sorting.C
```

File path and old file name to change — New file name

The following example will rename the directory **Temp** in the **D:** drive to **Backup.**

```
Example:  C:\> REN D:\Temp  Backup
```

Old directory name to change — New directory name



DEL / ERASE

- **DEL and ERASE**

The **DEL** command is used **to delete a file.** The **DEL** command is to be followed by the path and file name to delete as: **DEL [File_Path_And_Name]**

The following example will delete the file with the file name **Useless.txt.**

```
Example:  C:\> DEL Useless.txt
```

The following example deletes the file with file name **Prog15.C** located under **CProgs** folder under **D:** drive.

```
Example:  C:\> DEL D:\CProgs\Prog15.C
```

The **ERASE** command can also be used in place of the **DEL** command to delete a file as shown below.

```
Example:  C:\MyDocs\Word>ERASE letter32.doc
```



VOL

- **VOL**

The **VOL** command **displays the disk or drive space specified.** The command is typed as: **VOL [Drive]**

The following example will show the volume of the **C:** drive.

```
Example:  C:\> VOL C:
```

```
Output:   Volume in drive C has no label.
          Volume Serial Number is 7CB5-93F2
```



LABEL

- **LABEL**

The **LABEL** command in DOS is used **to create, change or delete the volume label** name of any drive. The command should be typed as: **LABEL [Drive_Name] [Label_Name]**

The following example will put the label name **Mydisk** for the **D:** drive.

```
Example:  C:\> LABEL D: Mydisk
```

- **Various External DOS Commands**

This section discusses the various important and common external DOS commands used.



DISKCOPY

- **DISKCOPY**

The **DISKCOPY** command is used to **copy the complete contents of one floppy disk to another.** The two floppy disks must be of the same type. The command is to be followed by the source and target drive names as: **DISKCOPY [Drive1]: [Drive2]:**

The following command will copy the contents of the floppy in drive **A:** to the floppy in drive **B:**

```
Example:  C:\> DISKCOPY A: B:
```

▪ TREE

TREE

This command allows the user to **view a listing of files and folders** in an easy to read **graphical format**, with the main directory at the top, followed by the sub and sub-sub directories along its branches. The format of the command is **TREE [Drive/Path]:**

```
C:\>tree c:\dir2
Folder PATH listing
Volume serial number is 7CB5-93F2
C:\DIR2
├───dir3
└───dir4
    └───dir5
```

The following command will display the directory tree structure of the directory named **dir2** under the **C:** drive.

Example: C:\> TREE C:\dir2

▪ MOVE

MOVE

The **MOVE** command can be used **to move a file from one location to another.** (Unlike a copy, by moving a file no copy of the file is kept in the source folder). It can also be used **to rename a directory.** After typing the command name, type the source file name along with the path and then type the destination path. The command is to be typed as: **MOVE [Source_File] [Target_File_Destination]**

The following example will move the file test.doc from the **C:** drive to the **D:** drive.

Example: C:\> MOVE test.doc D:

Source file name to move **Destination drive**

The following example will move the file project.doc from the sub-directory Word under the directory **MyDocs** in the **C:** drive to the directory **Temp** in the **D:** drive.

Example: C:\> MOVE C:\MyDocs\Word\project.doc D:\Temp

Source directory and file name to move **Destination directory**

The following example will move the same file **project.doc** from the sub-directory Word under **MyDocs** in the **C:** drive to the directory **Temp** in the **D:** drive, **but with the changed file name computer.doc.**

Example: C:\> MOVE C:\MyDocs\Word\project.doc D:\Temp\computer.doc

Source directory and file name to move **Destination directory with new file name**

The following example is used to **change or rename a directory** name from **MyDocs** to **Document.**

Example: C:\> MOVE MyDocs Document

Original directory name **Renamed directory name**

▪ XCOPY

XCOPY

The **XCOPY** command is used **to copy files and directory trees** from one location to another. The command will copy all the files and directories and sub-directories The command is to be typed as: **XCOPY [Source_Directory] [Target_Directory_Destination]**

The following example copies the directory structure under **C:\MyDocs** along with sub-directories and files to the **D:** drive.

Example: C:\> XCOPY C:\MyDocs D:

Source directory structure to copy **Destination drive**

▪ ATTRIB

ATTRIB

The **ATTRIB** command helps to **set the attribute for a file.** Different options available with **ATTRIB** are:

- +a Marks as an archive file i.e. the file can be edited if required; – a removes the attribute
- +r Marks the file read-only i.e. it can be opened but not edited; – r removes the attribute
- +s Makes the file a system file i.e. for files like io.sys, msdos.sys etc.; – s removes the attribute
- +h Makes the file hidden i.e. it is not displayed during a listing; – h removes the attribute

The following example will change the attribute of the file **MyData.txt** to a read-only, archive file.

Example: C:\> ATTRIB +r +a MyData.txt

The following example will remove the read-only attribute of the file **MyData.txt** by using **-r**.

Example: C:\> ATTRIB -r MyData.txt



BACKUP

- **BACKUP**

The **BACKUP** command in DOS is **used to take backup copies** of data, files, and directories from a hard disk to a required number of floppies. It is not possible to view the files directly from the floppies after backup. To get back the files in the normal format, the **RESTORE** command need to be used. The command should be typed as: **BACKUP [Source_Drive_Path] [Target_Drive_Path]**

Certain **switches** are used to facilitate backup. These include:

/s	Used to backup sub-directories
/m	Backs-up files which have been changed since last backup
/d:date	Backs-up only those files that have been modified after the specified date
/t:time	Backs-up only those files that have been modified after the specified time

The following example takes backup of entire **D:** drive into floppies of drive **B:**. The backup will include all sub-directories under the **D:** drive also.

Example: C:\> BACKUP D:\ B:/s



RESTORE

- **RESTORE**

The **RESTORE** command in DOS is **used to get back the files,** directories and sub-directories backed up using **BACKUP** command. The command name should be followed by the drive and path which contains the backup files and then the target drive as: **RESTORE [Source_Drive] [Target_Drive]**

Certain **switches** are used to facilitate restore operation. These include:

/s	Used to restore sub-directories
/p	Prompts for permission to restore
/b:date	Restores only those files that have been last modified on or before the specified date
/a:date	Restores only those files that have been last modified on or after the specified date
/e:time	Restores only those files that have been last modified on or before the specified time
/l:time	Restores only those files that have been last modified on or after the specified time
/m	Restores only those files that have been modified since the last backup
/n	Restores only those files that are non-existing in the drive

The following example restores all files from **B:** drive to the **D:** drive including sub-directories and will ask for permission with a prompt.

Example: C:\> RESTORE B:\ D:/s/p



UNDELETE

- **UNDELETE**

The **UNDELETE** command in DOS is **used to restore a deleted file,** if possible. The command should be typed as: **UNDELETE [Source_Drive_Path]**

Certain **switches** are used to facilitate backup. These include:

/dos	Recovers files listed as deleted by MS-DOS
/dt	Recovers files protected by Delete Tracker
/ds	Recovers files protected by Delete Sentry

The following example will undelete i.e. restore the file **project.doc** from the **D:** drive.

Example: C:\> UNDELETE C:\MyDocs\Word\project.doc



CHKDSK

- **CHKDSK**

The **CHKDSK** command in DOS **checks a disk for errors and displays a status report**. The command should be typed as: **CHKDSK [Drive_Name]**

Certain **switches** are used with this command. These include:

/f	Fixes errors on the disk
/v	Displays the full path and name of every file on the disk

The following example will check the **D:** drive for errors and fix them if possible.

Example: C:\> CHKDSK D:/f

- **SCANDISK**

SCANDISK

The **SCANDISK** command is used **to check a data disk for logical and physical errors** and repair the problems on the disk. File or folder errors can occur if the computer is not properly shut down. The scandisk utility checks the disk surface for any physical damage. It tries to repair the disk surface first. In case it is unable to do so, it marks the damaged area as containing bad sectors in the FAT and prevents further writing of data on that portion. It also checks and marks different files and folders for errors. The command should be typed as: **SCANDISK [Drive_Name]**

Certain switches are used with this command. These include:

/all	Checks and repairs all the drives
/autofix	Fixes the problems without prompting
/surface	Performs a surface scan after other checks

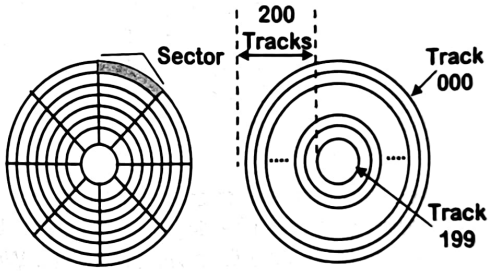
The following example scans all the drives.

Example: C:\> SCANDISK/all

- **FORMAT**

FORMAT

The **FORMAT** command in DOS is used **to lay down the pattern of tracks and sectors** onto a secondary storage device.



Tracks and Sectors

Before a secondary storage device can be used in a computer system it must be prepared by a process called disk formatting. When the command to format a disk is given (e.g. FORMAT A:), the disk drive's read/write head lays down a magnetic pattern on the disk surface. The pattern enables the disk drive to **organise and store data properly.**

Formatting first divides the surface of a disk into a number of invisible concentric circles called **tracks.** As shown in the figure, the tracks are numbered consecutively from the outermost to the innermost. Large capacity disks may contain up to a thousand tracks. Each track is again further subdivided into smaller sections called **sectors.** It is the smallest storage unit with which the disk can work i.e. disk drives are designed to read/write whole sectors at a time. Typically a sector contains 512 bytes. Each sector of a disk has a unique number called the disk address. An OS like DOS again combines up to 64 sectors to form a **cluster.** Each cluster forms the basic unit of memory read/write. The command should be typed as: **FORMAT [Drive_Name]**

Certain switches are used with this command. These include:

/v	Specifies the volume label
/q	Performs a quick format
/u	Performs an unconditional format
/b	Allocates space on the formatted disk for system files
/s	Copies system files to the formatted disk
/c	Test clusters that are currently marked as bad

The following example **quick formats** the **D:** drive.

Example: C:\> FORMAT D:/q

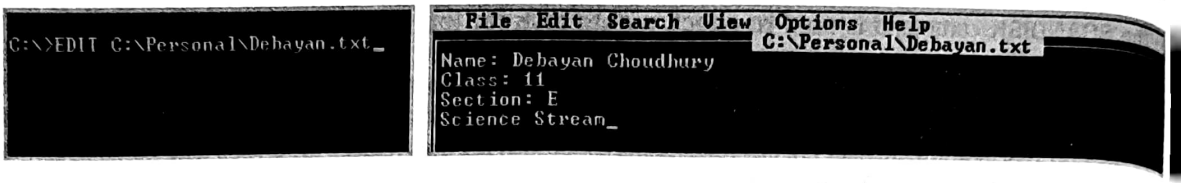
EDIT / EDLIN

▪ EDIT and EDLIN

The **EDIT** or the **EDLIN** command in DOS is used **to create and edit a data file**. The **COPY CON** command can also be used to create a data file, however the same command cannot be used to edit the file so created. Using the **EDIT** or **EDLIN** command one can both create and edit a data file. The command should be typed as: **EDIT [File_Name] or EDLIN [File_Name]**

The following example creates a file called **Debayan.txt** under the **Personal** folder in **C:** drive.

Example: C:\> EDIT C:\Personal\Debayan.txt



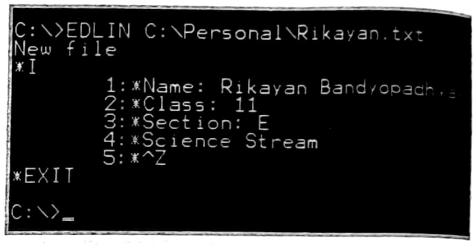
The blank **EDIT** window opens when the command is typed. Type the necessary data in the text area of the opened window and after finishing writing, go to the File menu and Save the file. The file **Debayan.txt** so created can also be modified or edited by simply retyping the above command, when the text file created earlier will again open. After editing the file it can be again saved in the same manner to incorporate the changes.

EDLIN is a line oriented text editor that can be used to create and edit a text file. The following example creates a file called **Rikayan.txt** under the **Personal** folder in **C:** drive.

Example: C:\> EDLIN C:\Personal\Rikayan.txt

Output:

```
New file
*I
    1:*Name Rikayan Bandyopadhyay
    2:*Class: 11
    3:*Section: E
    4:*Science Stream
    5:*^Z
*EXIT
```



When the command to create the text file is typed and the Enter key is pressed, then the message "New File" followed by a * will be displayed. Type "I" beside the * to insert new lines into the file. The Enter key can be used to create new lines within the file. When finished typing, press the F6 key or **Ctrl+Z** to come out from the Insert mode. Type **EXIT** to save the file and exit. To edit the file to add more data to the file, type the same command again and follow the same set of steps.

WILD CARD characters

▪ Wild Card characters * and ?

Two symbols, called wildcards, allow the user in DOS **to specify a group of files**. The wild card character '*' is used to indicate any group of characters while each '?' indicates a single character.

The following **COPY** command can be used **to copy all files with the file extension** doc, irrespective of their name, from the **C:** drive to the Temp directory also located under the **C:** drive.

Example: C:\> COPY *.doc C:\Temp

The following **DEL** command can be used **to delete all files with the file name Project**, irrespective of their extensions, from the **Temp** directory located under the **C:** drive.

Example: C:\> DEL C:\Temp\Project.*

The following **DIR** command can be used to **display all files** whose name starts with **Prog** (like **Prog12**, **Program143** etc.) and which have a single letter extension name (like **.C**) from the **D:** drive.

Example: C:\> DIR D:\Prog*.?

• Creating a batch file in DOS

Creating Batch File

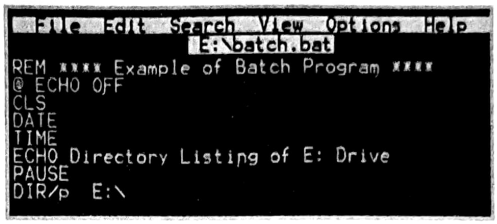
To wait for a command to complete execution before you can give the next command can be sometimes time consuming. But in applications where such interaction is not required as in applications that calculate employee salaries, one can use batch processing whereby DOS allows to group several commands into a file with the extension 'bat' (for batch). After saving the file, **simply type the name of the file to run the commands** one by one from the batch file.

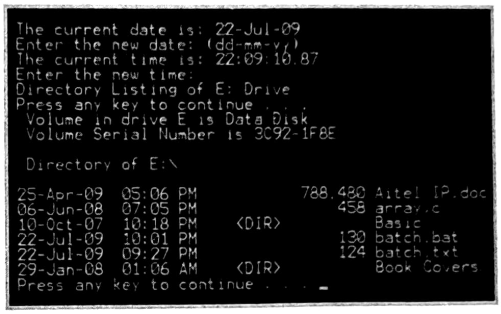
The following example can be used to create a batch file called batch.bat in DOS. The file is used to first clear the screen using the **CLS** command and then set the **DATE** and **TIME**. Finally the directory listing of the **E** drive is displayed in a page-wise manner using the **DIR/p** command. The batch file is created using the **EDIT** command as shown below.

E:\> EDIT batch.bat

Output:

```
REM **** Example of Batch Program ****
@ ECHO OFF
CLS
DATE
TIME
ECHO Directory Listing of E: Drive
PAUSE
DIR/p  E:\
```





In the above batch program the **REM** command is used to insert a comment in the file. Next **ECHO OFF** command is used to suppress the display of command names when these are run. The '@' symbol is used to suppress the display of the **ECHO OFF** command itself. The **ECHO** command is used to display the text following it onto the screen. The **PAUSE** command is used to pause for a key press.

Important DOS commands at a glance

DOS commands at a glance

Command	Use	Format / Example
ATTRIB	It helps **to set or reset the attribute for a file** as read only, archive, hidden or as a system file	**C:\>** ATTRIB +r +a MyData.doc **C:\>** ATTRIB -r MyData.doc **C:\>** ATTRIB +h *.sys
BACKUP	It is **used to take backup copies of one or more files** from one disk to another	**C:\>** BACKUP D:\ B:/s
CHDIR or **CD**	It is used **to change from one directory to another** in DOS. Along with the command, the path and name of the directory to move to is to be typed	**C:\>** CD C:\Personal\Letter **C:\Personal\Letter>** CD.. **C:\Personal\Letter>** CD\
CHKDSK	It is used to **check a disk for errors and displays a status report**	**C:\>** CHKDSK D:/f
COPY	It is used **to copy a file** from one location to a different one or make a copy of a file with a different name at the same location. The copy command is to be followed by the source and the destination file names and file paths	**C:\>** COPY file1.txt backup.txt **C:\Personal>** COPY file2.c D:\ **C:\>** COPY file3.c C:\Personal **C:\>** COPY *.bmp C:\Personal
DATE	It is used to **display or modify the current date.** After typing the command, type a new date in the format dd-mm-yy or press Enter to keep the same date	**C:\>** DATE
DEL and **ERASE**	It is used **to delete a file.** The del command is to be followed by the path and file name to delete	**C:\>** DEL prog.c **C:\>** DEL C:\Home\Progs\Try.c **C:\>** ERASE D:\Temp\pic.bmp
DIR	It is used to **list all the files or sub directories** under a given directory. Along with the file name, the date & time of file creation and the size of the file is also displayed	**C:\>** DIR **C:\>** DIR/p **C:\>** DIR personal
DISKCOPY	It is used to **copy the complete contents of one floppy disk to another**	**C:\>** DISKCOPY A: B:

Command	Use	Format / Example
EDIT and EDLIN	These commands can be used **to create and edit a text file**	**C:\>** EDIT C:\Debayan.txt **C:\>** EDLIN C:\ Rikayan.txt
FORMAT	The FORMAT command In DOS is used **to lay down the pattern of tracks and sectors** onto a secondary storage device	**C:\>** FORMAT D:/q
LABEL	It is used **to create, change or delete the volume label** name of any drive	**C:\>** LABEL D: Mydisk
MKDIR or MD	It is used **to make a new directory** or folder in DOS. Along with the command, the path i.e. the location and name of the new directory is also to be typed	**C:\>** MD Project **C:\>** MD C:\Personal\Letter **C:\>** MKDIR D:\Temp
MOVE	It can be used **to move a file from one location to another**. It can also be used **to rename a directory**. After typing the command name, type the source file name along with the path and then type the destination path. In case you want to change the file name after moving the file, then write the new file name	**C:\>** MOVE file1.txt D:\Temp **C:\>** MOVE bd.c D:\Temp\bak.c **C:\>** MOVE C:\Temp\prg.c D:\ **C:\>** MOVE *.txt D:\Temp
RENAME or REN	It is used **to rename a file or a directory**. The rename command is to be followed by the path and file/directory name to be renamed and the changed file/directory name	**C:\>** RENAME prog.c final.c **C:\Personal>** REN TC.exe C.exe **C:\>** REN Personal Home
RESTORE	It is **used to get back the files**, directories and sub-directories backed up using BACKUP command. The command name is followed by the drive and path which contains the backup files and then the target drive	**C:\>** RESTORE B:\ D:/s/p
RMDIR or RD	It is used **to remove or delete a directory** or folder in DOS. Along with the command, the path i.e. the location and name of the directory to be removed is to be typed	**C:\>** RD Project **C:\>** RD C:\Personal\Letter **C:\>** RMDIR D:\Temp
SCANDISK	The SCANDISK command in DOS is used **to check the hard disk or floppy disks for logical and physical errors** and repair the problems on the disk	**C:\>** SCANDISK/all
TIME	It is used to **display or modify the current time**. After typing the command, type a new time in the format hour-min-sec or press Enter to keep the same time	**C:\>** TIME
TREE	Allows the user to view a listing of files and folders in an easy to read graphical format, with the main directory at the top, followed by the sub and sub-sub directories along its branches	**C:\>** TREE C:\Personal
UNDELETE	It is **used to restore a deleted file**, if possible	**C:\>** UNDELETE D:\project.doc
VOL	It is used **to display the disk or drive space specified**	**C:\>** VOL C:
XCOPY	It is used **to copy files and directory trees** from one location to another. The command will copy all the files and directories and sub-directories under it	**C:\>** XCOPY C:\Temp D:\Home
Wild Card characters * and ?	Two symbols, called wildcards, allow the user in DOS **to specify a groups of files**. The wild card character '*' indicates any group of characters while '?' indicates a single character	**C:\>** COPY *.doc C:\Temp\ **C:\>** DIR D:\Prog*.?



Memory Management

6.4 Windows Operating System

Microsoft Corporation, co-founded by Bill Gates (picture on the right) introduced the **first version of the Windows Operating System in 1985**. Unlike MS-DOS, Windows used a GUI and allowed users to run multiple applications at the same time using multitasking features. It also allowed running of application programs that required memory larger than the available memory using the virtual memory concept.

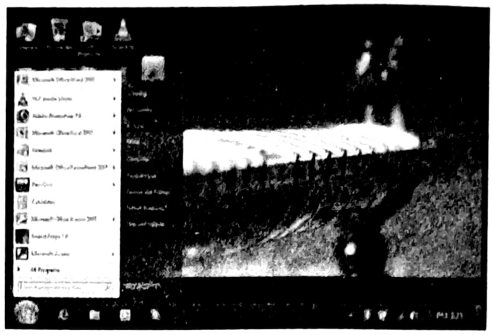


A Brief History of the Windows OS

The first version was **Windows 1.0** that was released in November 1985. This was followed by **Windows release 2.0, 3.0,** and **3.1** released in 1992. The next advancement was **Windows 95** which was released in 1995. It had a new user interface and provided various new features. Microsoft next released **Windows 98** in 1998. In 2000 Microsoft released **Windows-Me** (**M**illennium **E**dition). The next step taken by Microsoft was to combine the features of their consumer and business operating systems. This led to the development of **Windows XP**, which was available in both home and professional versions around 2002.

The first Operating System Microsoft coded was **Xenix**, which was a version of UNIX.

The next release was **Windows Vista** in 2007. New features of Windows Vista included an updated graphical user interface and a **visual style called Aero**, and a new search component called Windows Search. The main objective with Windows Vista was to make improvements on the security features of the operating system.

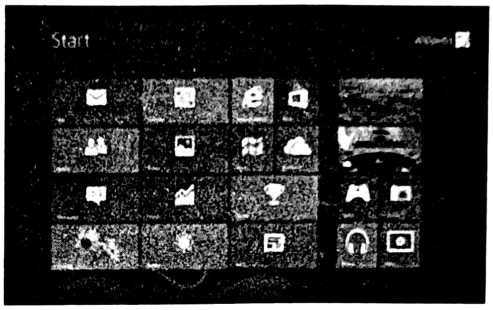


Windows 7 with Start menu

Windows 7 is available in six different editions, of which the Home Premium, Professional, and Ultimate are most common

Windows7 was released in 2009. It was basically an upgrade, designed to work with Vista-compatible applications and hardware. Some standard applications that have been included with prior releases of Microsoft Windows, like Windows Calendar, Windows Movie Maker, Windows Photo Gallery, etc. are not included in Windows7. These are instead available separately free of cost, as part of the **Windows Essentials suite.**

The latest operating system released by Microsoft for use with home, business desktops, and portable computers till date (as of 2013) is **Windows8**. It was released in the year 2012. It is primarily **designed for use with mobile devices** such as tablets, smart phones etc. Windows8 features a new **touch user interface** with a new **Start screen that replaces the Start menu** of earlier Windows versions.



Windows 8 with Start screen



Windows1 1985



Windows 3.1 1992



Windows 95 1995



Windows XP 2001



Windows Vista 2006



Windows 7 2009



Windows 8 2012

Differences between DOS and Windows

DOS	Windows
1. DOS has a Character User Interface (CUI) that some people find difficult to run as all commands need to be memorised	1. Windows has a Graphical User Interface (GUI), making it easier for a new user to learn and use the system as commands need not be memorised
2. Different screen modes used for graphical and textual data	2. Graphical and textual data can be represented under one screen mode
3. It restricts the user to eight character file names with three character extensions	3. It allows filenames up to 255 characters long including blank and certain punctuation marks
4. DOS was mainly designed for use with 16-bit CPUs	4. It can make use of the 32-bit CPUs that are faster in operation
5. DOS was designed to handle programs whose maximum size was 640 Kb	5. Windows can run programs with size greater than 640 Kb
6. It is a single-user, single-task operating system i.e. a user can run only one program at a time	6. It is a multi-user multitask operating system i.e. a user can run more than one program at a time
7. Programs operating under DOS usually have non-standard features, i.e. the same set of operations like copy and paste can be done in different ways, making it difficult for the user	7. Windows programs conform to a standard way of working. For example a MS Windows word processor program works similar to the way a MS Windows spreadsheet program
8. Use of mouse and drag an drop features are usually absent. This makes the execution of certain commands lengthy or difficult	8. Drag and drop features make certain command execution easier (like dragging a file from one folder to another to copy it)
9. Plug and play options are not available	9. Features plug and play options (from Win-95)
10. Object Linking and Embedding (OLE) feature not available	10. OLE helps to embed the output of an application into another application with automatic updates

Components of Windows system

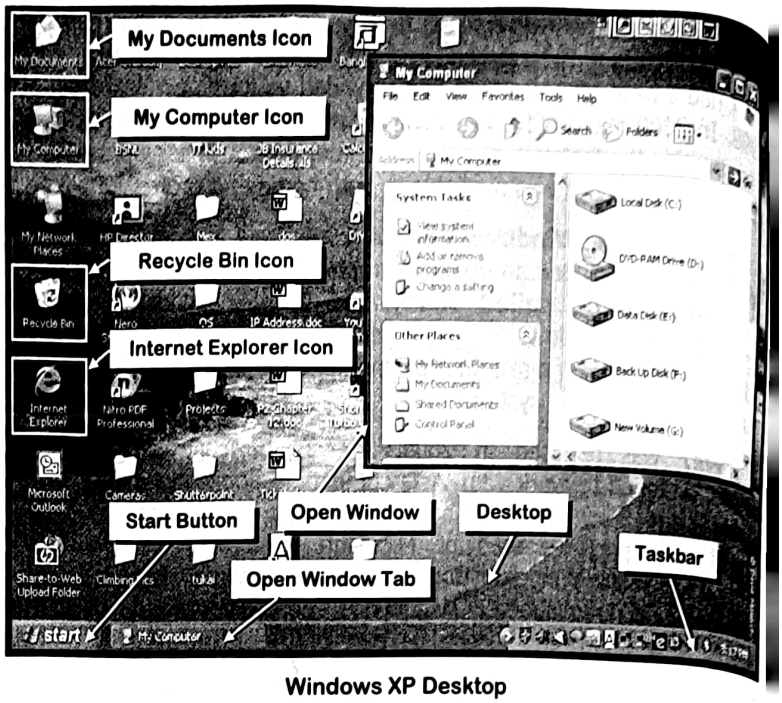
Components of a Windows System

Unlike DOS, Windows has various **graphical components** that are used for specific operations in the Windows environment as stated below:

Bill Gates was ready to launch Windows under the name 'Interface Manager' before he was persuaded by an employee to change it

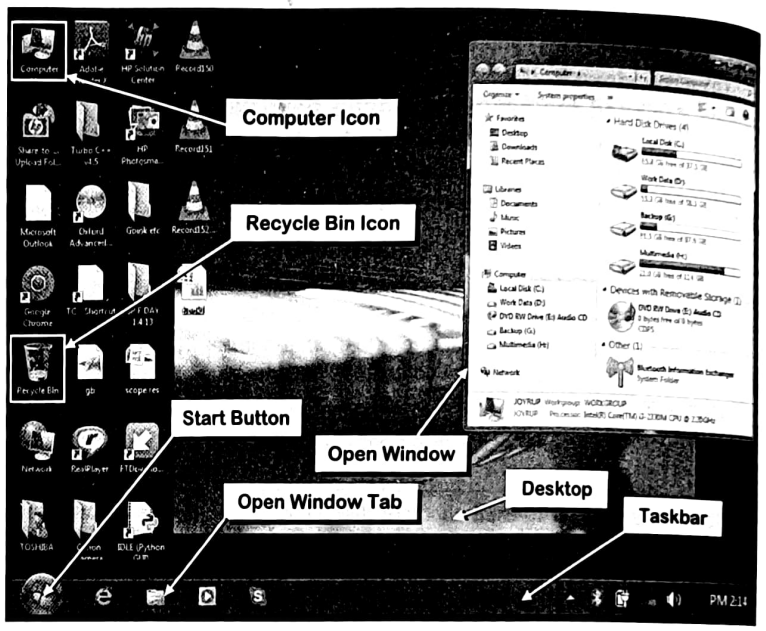
Press the Win key + Space bar on the keyboard to preview the Desktop anytime

- **Desktop and Icons**: The working space of the Windows system is known as the *Desktop*. Over the desktop are placed **small pictures called icons** of the different parts of **the computer that one works** with like files, folders, recycle bin, network etc. The icons can be controlled by using the mouse and its on-screen pointer. An icon can be activated by double clicking on it using the mouse.



Windows XP Desktop

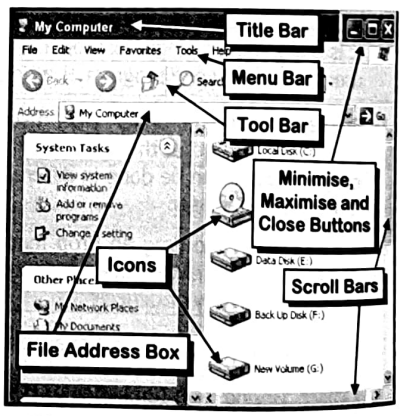
- **Taskbar**: At the bottom of the desktop is an area in the form of a band, called the Taskbar. On the leftmost part of the taskbar is a button called the **Start button** (present in all versions of Windows from Win95, up to Windows7. Not present in Windows8). After one clicks on the Start button, it displays **different dropdown lists**. One can click on the program icons that are displayed by the dropdown lists and run them. Whenever a program starts, an **open window tab** for it appears on the Taskbar. When multiple programs are running, one can shift from one program to another by clicking on these buttons on the taskbar. The taskbar also displays the **system time and program icons for quick launch** of programs by a single click on them.



Windows7 Desktop

- **Window**: When a program icon is clicked and the program is loaded into memory, the **program appears in a rectangular frame called a Window** on the screen. By manipulating these windows on the screen one can work with multiple programs that have been loaded into the memory.

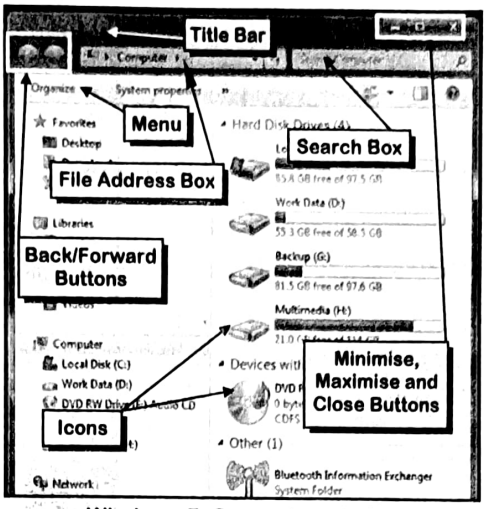
- **Title Bar**: Each window contains a Title Bar across the top that indicates **what the window contains**. When a particular window is chosen, the title bar of that window gets highlighted. The entire window can be moved by clicking on the title bar and dragging it to any desired location. The title bar also contains three buttons for manipulating the windows. These are the **Minimise**, **Maximise** and **Close buttons**. On clicking the



Windows XP, My Computer window

Maximise button, the program is displayed occupying the entire screen. On clicking the Minimise button, the program becomes inactive and is displayed as a button on the Taskbar. The program quits on clicking the Close button.

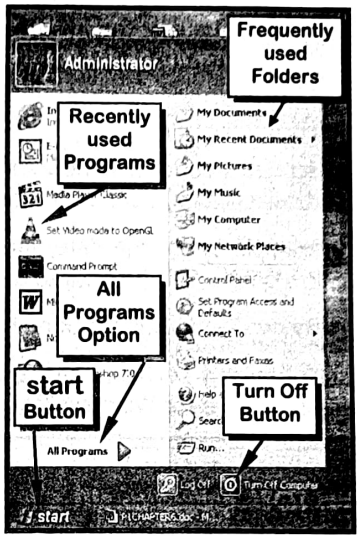
- **Menu Bar & Toolbar**: The region below the title bar represents the actual working area of the program. It contains **a horizontal list of menus called the Menu bar**. Below the Menu bar is a **set of buttons that form the Toolbar**. The toolbar is used to give different instructions by clicking on the buttons. **Dropdown lists** appear when the user clicks on a menu item. From the menu choose an option like opening, saving, or closing a file or any other instruction to use.
- **Scroll Bar**: The window can also contain Scroll Bars to view the different parts of a program or file that are longer or wider than the size of the window.



Windows 7, Computer window

The Windows Registry

The registry is the place where Windows **keeps most of its configuration information**. Windows needs the registry to boot. Therefore each time there is a successful booting it saves the contents of the registry as the **last known good configuration**. To prevent malfunction in case the registry gets damaged, Windows maintains a system component called **system restore**. This feature periodically saves the different configuration files and device driver states. In case the system boots but does not function as expected then it can be **restored to a previous working state** using this feature.



Taskbar in Windows XP

Command Line Interface

Working with Windows

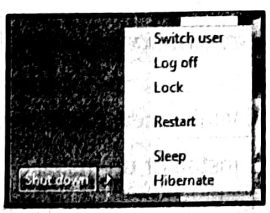
The user is presented with the **Windows Desktop** after the computer is switched on and Windows boots up. On the *Desktop* are the icons for *My Computer* (*Computer* in Windows7), *My Documents*, and *Recycle Bin*. There can be icons for other folders/files placed by the user. The **Windows Taskbar** is also visible at the bottom.

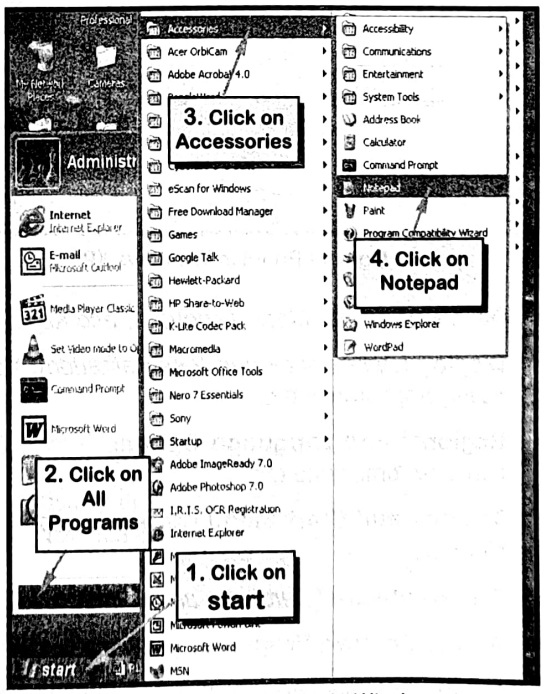
Working of Windows

To access any file or folder, click on the *My Computer* (*Computer* for Windows7) icon on the *Desktop*, when the *My Computer* window opens. It displays the **different drives** in the computer (like the hard disk, DVD drive etc.) as icons. By double clicking on an icon, open the required drive to access the required folder or file.

When a program is installed, an entry for that program is made under the *All Programs* dropdown list. To run the program, click on the *Start* button at the left bottom corner of the *Desktop* screen. A dropdown list displays the *All Programs* icon, a list of frequently used programs and folders, the **Turn Off Computer** button (**Shut down** in Windows7) and some other utility program icons.

In Windows XP click on the ***Turn Off Computer*** button to switch off the computer. In Windows7, do the same thing by clicking on the ***Shut down*** button from the start dropdown list. In Windows7, the Shut down option opens a dropdown list with various options as shown above.



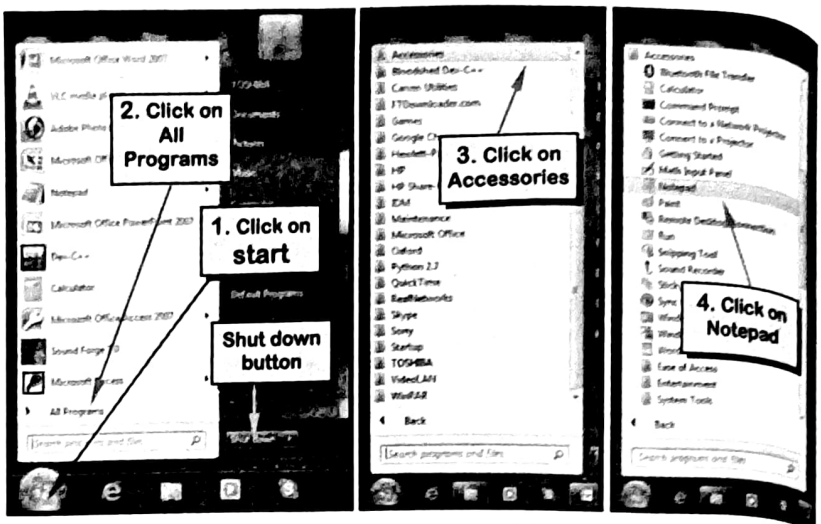


Steps to open a program in Windows XP

Press Alt+F4 from the keyboard to close an open window.
Double click on the title bar to maximize a window

To run a program, first click on the start button on the Taskbar and follow the various dropdown lists. If, for example you want to run the *Notepad* text editing program, then click on *Start → All Programs → Accessories → Notepad*

Refer to the diagram in the last page and on the right for the process of opening a program under Windows XP and Windows7.



Steps to open a program in Windows7

Press the Win key on the keyboard to open the Start menu

Changing System settings

• Changing System Settings

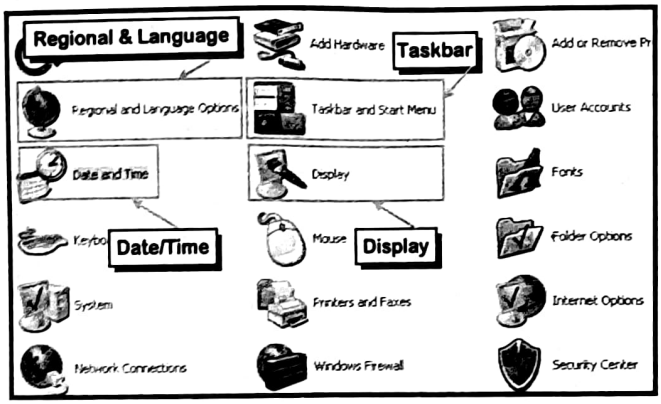
Like in DOS, a user can change the system time, date, display settings, regional settings, and other settings. To make these changes click on: Start → Control Panel

The screen shots below show some of the **options available under the control panel** window. To access a particular option, double click on the required icon to open the window to make changes to that option.

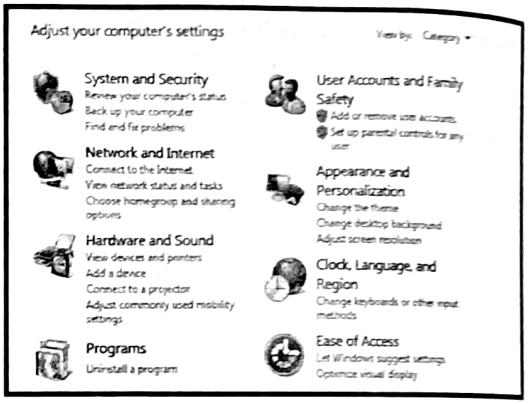
For example to change date/time, double click on *Date and Time* icon, when the *Date and Time Properties* window opens. Set the desired date and time and then press the *Apply* button to apply the changes.

In Windows7, the same is done by clicking the *Clock, Language, and Region* option.

The **following settings can be done from the control panel in Windows XP** (the options within the parenthesis are for Windows7):



Control Panel in Windows XP

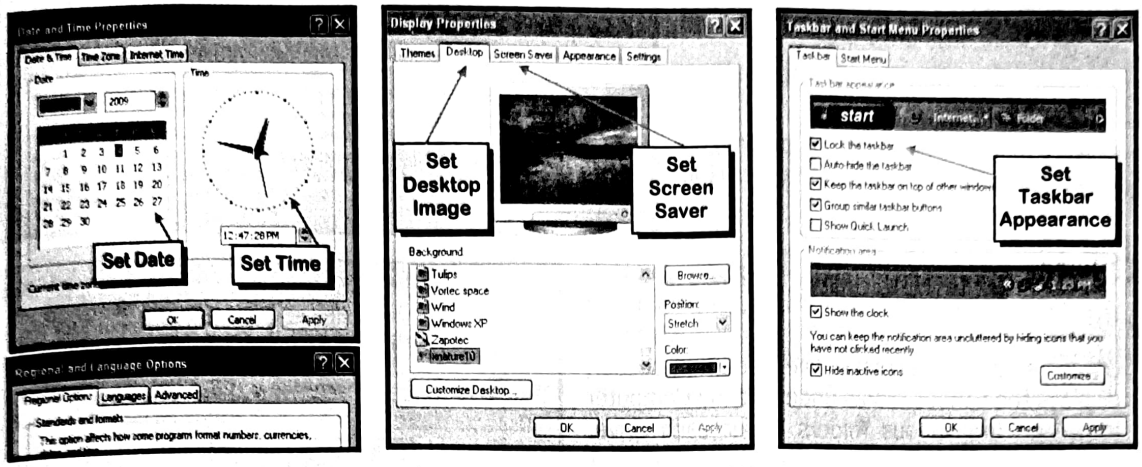


Control Panel in Windows7

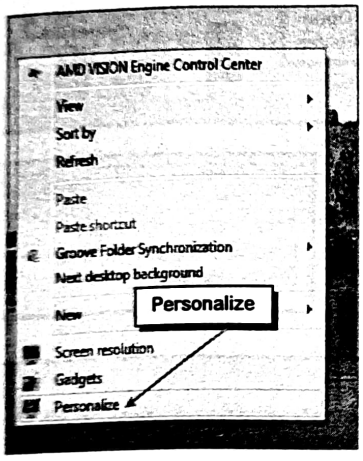
- **Date and Time** (*Clock, Language, and Region*): Used to change the system date and time.
- **Display** (*Appearance and Personalization*): Used to change display settings like Desktop Picture, Screen Saver, Appearance etc.
- **Regional and Language Options** (*Clock, Language, and Region*): Used to specify regional language, currency, time zone etc.
- **Taskbar and Start Menu** (*Appearance and Personalization*): Used to hide/display taskbar, show system clock etc.
- **Add Hardware** (*Hardware and Sound*): Used to install a new hardware.
- **Add or Remove Programs** (*Programs*): Used to install or un-install a particular program.
- **Folder Options** (*Appearance and Personalization*): Used to change folder settings
- **Fonts** (*Appearance and Personalization*): Used to install or un-install a particular font

- **Keyboard** (*Hardware and Sound*): Used to change settings like key repeat rate, blink rate etc.
- **Mouse** (*Hardware and Sound*): Used to change mouse settings like button configuration, speed etc.
- **Printers and Faxes** (Hardware and Sound): Used to install a new printer or fax machine
- **Internet Options** (*Network and Internet*): Used to set various options to access the Internet.
- **System** (*System and Security*): Used to set Computer Name, Hardware, System Restore option etc.
- **User Accounts** (*User Accounts and Family Safety*): Used to set administrative and user accounts.

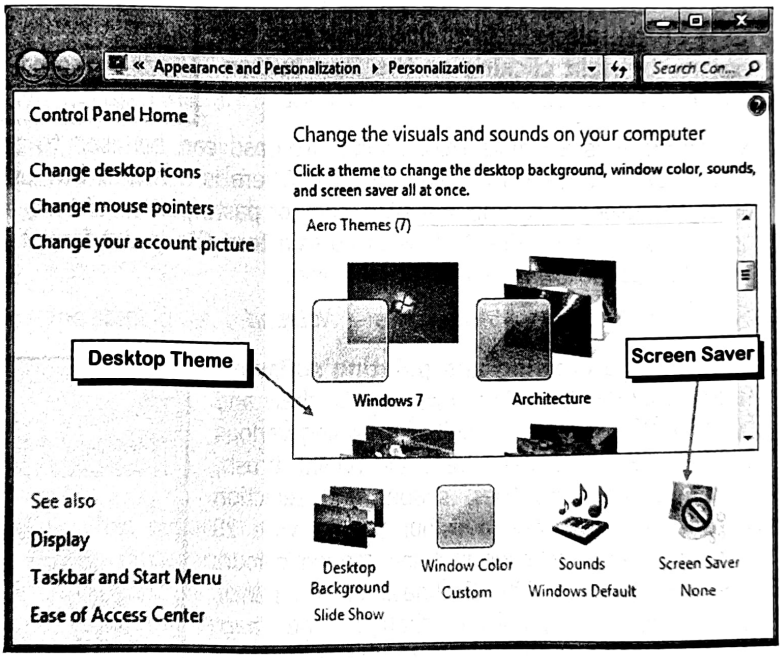
System Settings



Settings options under Windows XP



In Windows 7, right click on the desktop and from the dropdown list select the **Personalize** option to open the dialogue box to change desktop settings and apply a screen saver.



Settings options under Windows 7

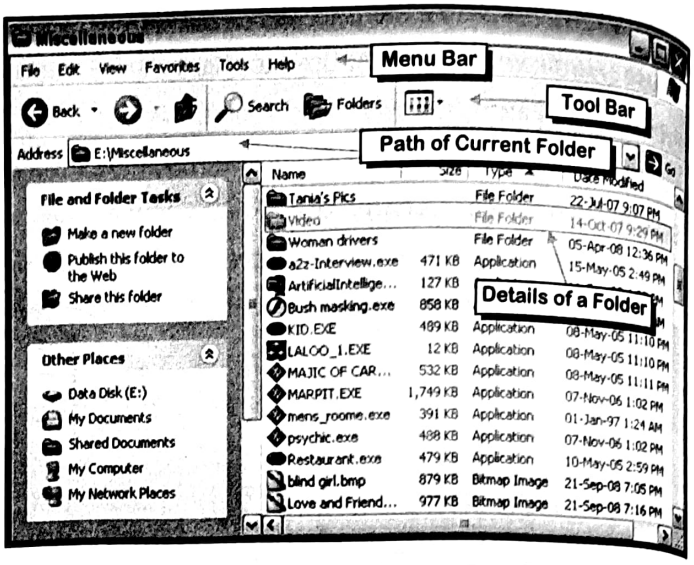
- **Various Utilities Provided by Windows**

Windows provides various utilities related to file and folder related jobs as discussed below:

1. **My Computer** (called *Computer* in Windows7): The *My Computer* or *Computer* option is used to do all types of file and folder operations. It is available on the *Desktop* and also under the *Start* menu. The *My Computer* window along with its menu bar and tool bar is shown in the next page. The Address box displays the path of the currently selected file or folder and the title bar shows the selected folder name. The different operations possible using *My Computer* are:
   - See the folder structure and contents of the different drives like the hard disk, floppy disk, CD/DVD drive, flash drive etc.

Press Win key + Home key to minimize all windows except active window. Repeat process to bring them back

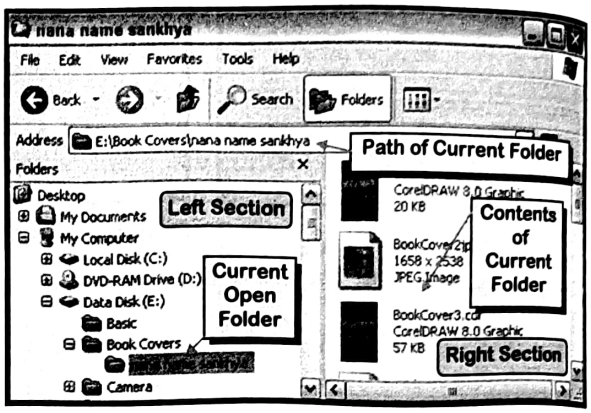
- Copy/move folders and files from one location to another.
- Rename/delete folders and files.
- See and set the properties of files.
- Run programs or open files by double clicking on them or pressing the enter key after selecting t hem.
- Format secondary storage media like hard disk, flash drive etc.
- Open the *Recycle Bin* to restore deleted files.
- View and access a list of networked computers, in case the computer is in a LAN.



Press the Win key + E from the keyboard to open the Windows Explorer / Computer

Windows Explorer

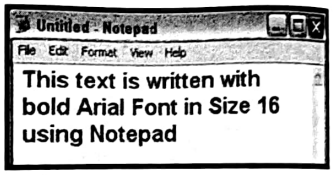
2. **Windows Explorer**: The *Windows Explorer* works similar to the *My Computer* option. The picture shows the explorer window which is divided into two sections. The left hand section displays the various drives under the My Computer icon, and the various folders under each drive. When a drive or folder is selected from the left section, the right section displays the contents of that selected folder. The title bar displays the name of the selected folder. One can access the explorer by **right clicking on the** Start **button** and selecting the Explore option from the list.



Notepad

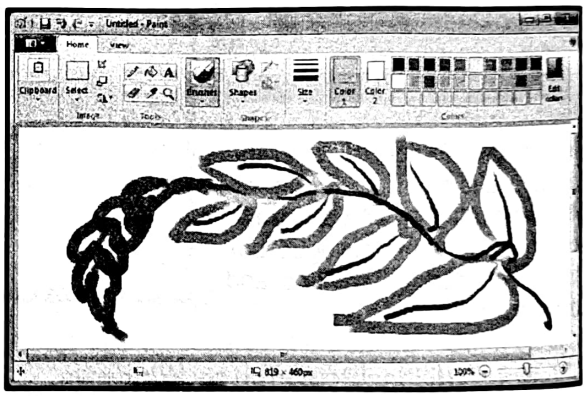
3. **Notepad**: It is a simple **text editor**. Notepad can be used to type simple text in a given font style and size. There is a **menu bar** using which the user can set the font style, copy or paste portion of text, and open, close or save a file. A file is saved as a **text file** in .txt format. To open *Notepad* follow the steps shown below:

   Start → All Programs → Accessories → Notepad



Paint

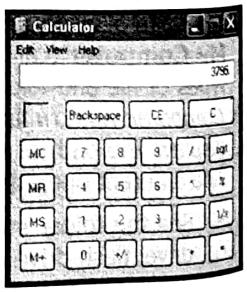
4. **Paint**: It is a **drawing and painting software** supplied by Windows. It can be used to draw and colour pictures. It has a toolbar providing various drawing tools like paint can, air brush, brush, pencil, eraser, text, lines, shapes and selection tools. It also provides a colour palette with 28 different colours to choose. User defined colours can also be used. The thickness of the pencil, eraser, and brush can also be changed. The *Image Menu* provide s options to rotate, flip or stretch a picture and also get an inverted coloured image. A *Paint* file can be saved as a **bitmap image** i.e. in a .bmp format or as a .jpeg format.



   To open Paint follow the steps shown below:

   Start → All Programs → Accessories → Paint

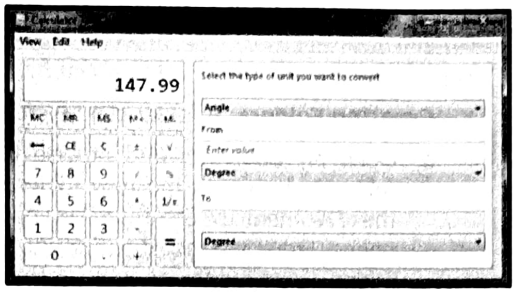
   The **Paint window** for Windows7 is shown above.



Calculator

5. **Calculator**: It is a **software calculator** supplied by Windows. It can be used to do any calculation just like a normal electronic calculator. The result of a calculation can be copied and used in another application also.

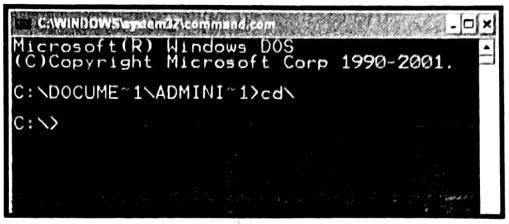
One can also use a **scientific calculator** to carry out scientific operations like trigonometric functions, logarithmic functions, exponential functions etc. Using the scientific calculator one can also work in **different number systems** like decimal, octal, binary, and hexadecimal. Different units of angles like degree, radian, and grad can also be used. Go to *View Menu* to open the scientific calculator.



To open the calculator follow the steps shown below (Calculator for WindowsXP & Windows7 are shown above)

Start → All Programs → Accessories → Calculator

6. **Command Prompt**: This is basically the MS DOS prompt in the Windows environment, i.e. it provides a **command line shell for Windows XP**.

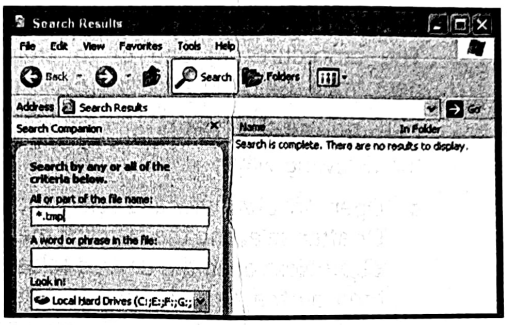



Command Prompt

It is provided by a special application called the **Virtual DOS Machine** (VDM). The VDM is based on MS DOS 5.0 source code. To open the command prompt, follow the steps shown below:

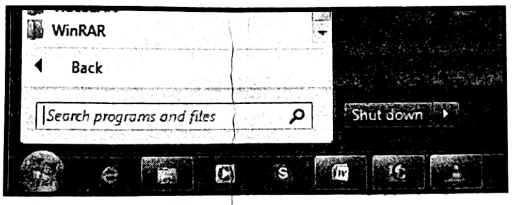
Start → All Programs → Accessories → Command Prompt

7. **Search**: This option is used to search for any type of file, folder, or computer (in case of a networked computer). After opening the *Search* window, the user has to type the name of the file to search and specify the drive name. In case a specific file name is not known, then a word or phrase in the file can be used as the search criteria.




Search

Moreover multiple files with a **common file extension** or **similar names** can also be searched using **wild card characters**. For example to search all files with the extension *.docx* type the file name as **.docs* using the wild card '*' to indicate any file name.

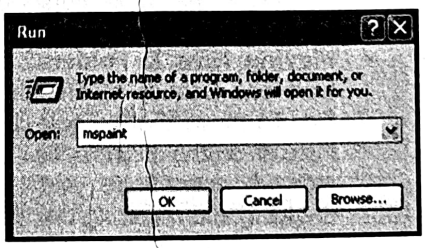


To open the search window follow the steps given below:

Start → Search

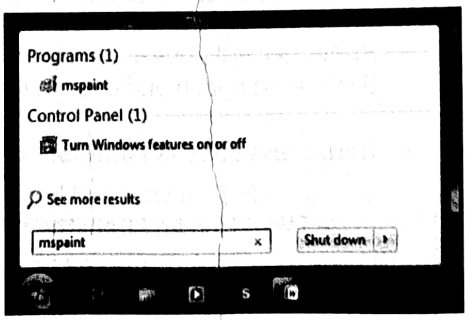
The *Search box* for **Windows7** is also shown. Here the *Search programs and files* option is available as a text box under the *Start* menu.

8. **Run/Search programs and files**: This option in Windows can be used **to open a program, folder, document or web site.** To run a program in WindowsXP go to *Start* and select *Run*. Next type the program name along with its path in the *Open* text box. One can even use the *Browse* option to locate the required file.




Run

However, for Windows7 the same *Search* option is used to run an application also. The screen shots on the right show the same in Windows XP and in Windows7. Some of the applications that can be run directly from the *Run* dialog box of Windows XP include:



- **Calculator**: Type *calc*
- **Command Prompt**: Type *cmd*
- **Paint**: Type *mspaint*
- **Notepad**: Type *notepad*
- **Windows Explorer**: Type *explorer*
- **My Documents Folder**: Type *my documents*
- **System Information**: Type *msinfo32*

File & Folder operations

• Various File & Folder Operations and Utilities in Windows

Let us now discuss the various file and folder operations possible using Windows.

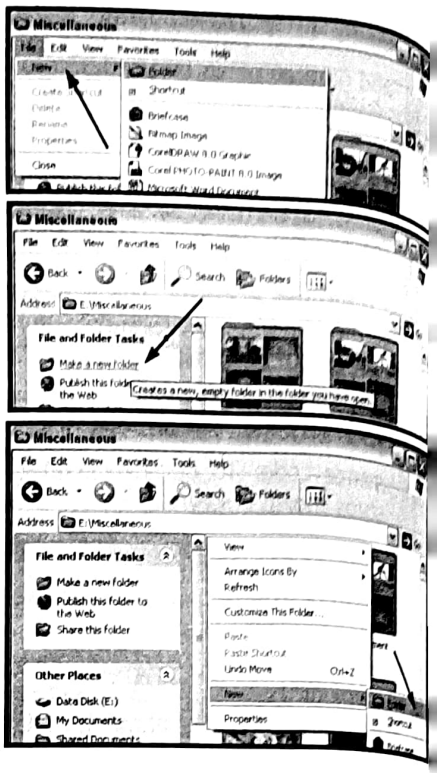
- **Creating Folders**: A folder can be created in the following ways:

  a. Open *My Computer* and browse to the desired drive or folder under which the new folder is to be created. From the *File* menu select the *New* option. A drop down list opens. From it select the *Folder* option to create the new folder. Ty pe the folder name in the text box provided.

  b. Open *My Computer* and browse to the desired drive or folder under which the new folder is to be created. Select the *Make a new folder* option from the left side section of the opened window. Type the folder name in the text box provided.

  c. Open *My Computer* and browse to the desired drive or folder under which the new folder is to be created. Right click on the right section of the opened window when a drop down list is displayed. From the list select the *New* option. Another drop down list opens. From it select the *Folder* option to create the new folder. Type the folder name in the text box provided. Folders can also be created on the *Desktop* using this procedure.

  The procedures a) and c) to create folders can also be performed using the *Windows Explorer*.



Shortcut: Press the Ctrl+Shift+N keys on the keyboard to create a new folder

You cannot create a folder and name it 'con' in Windows.

Copying Files & Folders

- **Copying Files & Folders**: A file or a folder can be copied in the following ways:

  a. Open *My Computer* and select the desired file(s) or folder(s). Click on the *Edit* menu and click on *Copy*. Or after selecting the file click the right mouse button to open a **drop-down menu**. From there select *Copy*. Next open the desired folder where the file needs to be copied. Again click on the *Edit* menu and then on the *Paste* button. Or after selecting the folder, click the right mouse button to open a drop-down menu. From there select *Paste*.

  b. Instead of opening *My Computer*, open *Windows Explorer* and repeat the same process as described for *My Computer*.

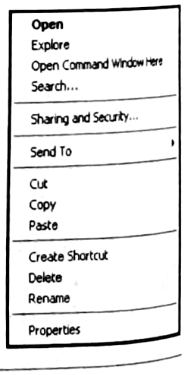
  c. Open *Windows Explorer*, find the folder and then the file(s) to copy. Select and **drag the file(s)** while pressing down the left mouse button and the *Ctrl* key on the keyboard and **drop the file(s)** by releasing the buttons after placing the mouse cursor over the destination folder.



Moving Files & Folders

- **Moving Files & Folders**: A file or a folder can be moved from one location to another in the same way as copying a file. To move the file **select the** *Cut* **option** from the menu instead of the *Copy* option. When using the drag-and-drop method **press the** *Shift* **key on the keyboard** while dragging and releasing the files to move.

A file can be copied or moved by **using another method** also. **Right click on the file** and drag it to the destination folder and release the mouse button. A **drop down list** will appear with the options to *copy* or *move* or *create a short-cut*. Select the desired option and click on it using the mouse.



Renaming Files & Folders

- **Renaming Files & Folders**: To rename a file or a folder:

  a. Open My Computer and browse to the desired file or folder. Right click on the file or folder icon and from the **drop down list** select the Rename option.

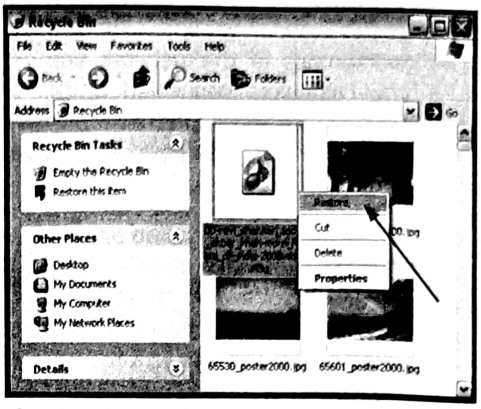
  b. Open My Computer and browse to the desired file or folder. After selecting the file or folder icon, press the F2 key from the keyboard to rename it.

- **Deleting Files and Folders**: A file or a folder can be deleted in the following ways:
  a. Open *My Computer* and browse to the desired file or folder. Right click on the file or folder icon when a **drop down list** is displayed. Select the *Delete* option from the list to rename the file or folder.
  b. Open *My Computer* and browse to the desired file or folder. After selecting the file or folder icon, press the *Del* or *Delete* key from the keyboard to delete the file or folder.

- **Recovering Deleted Files & Folders**: Files or folders deleted by a user are not permanently deleted from the system. These are first stored in a special folder called the *Recycle Bin*. Therefore a file or a folder can be recovered after deletion, by going to the *Recycle Bin*. To recover a file after deletion, first double click on the *Recycle Bin* icon situated on the *Desktop*. Then select the name of the deleted file from the recycle bin file listing, and click on the *Restore* option under the *File* menu. One can also right click on the file and click on the *Restore* option from the drop down list provided.







Recovering Deleted files

Press Shift+Del to delete a file without sending it to Recycle Bin

The *Empty* the *Recycle Bin* option shown at the left of the window can be used to remove permanently all files stored in the *Recycle Bin*. To delete a file permanently without sending it to the *Recycle Bin*, select the file and press the *Shift* and the *Del* keys together from the keyboard.

- **Disk Defragmenter**: This utility can be used to **analyse and defragment files** in a secondary storage like a hard disk. Over time files are stored and deleted from the hard disk. When a file gets deleted, the space for that file can be used to store another file. However, the existing space may not be sufficient for the new file. This leads to fragmented files, whereby the new file is saved partly in the deleted area and partly in another available area, with the two areas being linked. It takes **more time opening a fragmented file**. The disk defragmenter utility makes space to **store files again in adjacent locations**. To open this utility follow the steps:

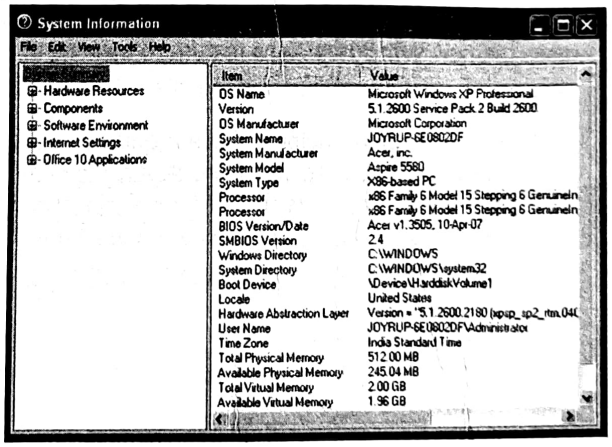
  Start → All Programs → Accessories → System Tools → Disk Defragmenter

Disk Defragmenter

- **System Information**: This utility in Windows can be used to **view various system information**. It opens a dialog box to display information regarding the Operating System name and version, the computer system model, the boot device, user name, the total and available physical memory, the amount of virtual memory allocated etc.

  To open the *System Information* option, follow the steps:

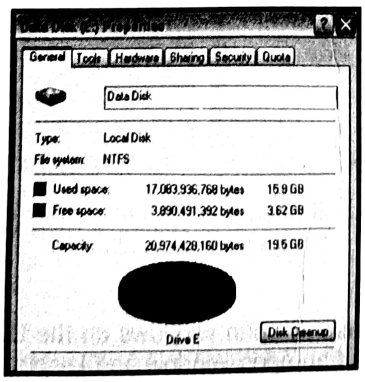
  Start → All Programs → Accessories → System Tools → System Information

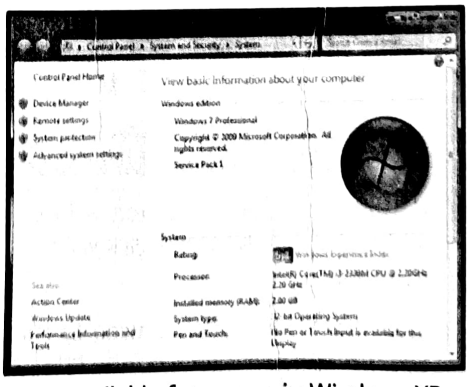


System Information

Press Win+Pause keys from the keyboard to view the System Information

- **View Disk Properties**: The properties of a secondary storage device like a hard disk, floppy disk, or flash drive can be viewed by opening the *My Computer* window and then right clicking on the desired drive icon. From the drop down list that is displayed, select the *Properties* option to open the properties window. It shows the total capacity of the drive, the used space, and available free space in Windows XP.



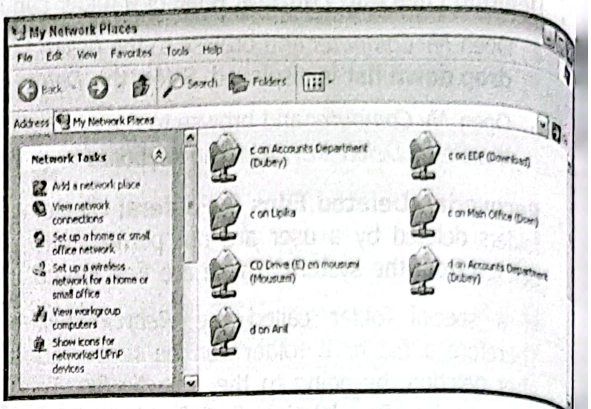


Disk Properties

- **My Network Places (Network):** This option, available on the *Desktop* can be used **to view the other computers and devices connected in a network**. It shows the various existing workgroups in the network and the computers connected under each workgroup. Other network resources like printers are also displayed in this window.



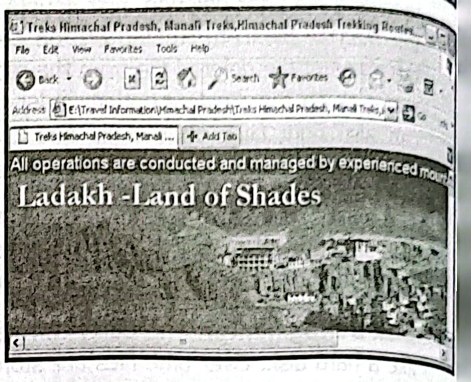




Internet Explorer

- **Internet Explorer:** The *Internet Explorer* is different from the *Windows Explorer* and is used to **browse the Internet**. It provides an *Address box* where the uniform resource locator of a web site to be searched in typed. Next the *Go* button is pressed to load the web page into the browser window. While viewing through a web page one **can use hyperlinks** to navigate to other web sites or can use the Back button to view any previously opened page. You can access the *Internet Explorer* from the *Desktop* or from the *Start* menu.





- **View Pictures:** To view pictures in various formats like .bmp, .jpg, .gif etc. one can use the *Windows Picture and Fax Viewer* option. To use it, simply right click on the picture file icon from *My Computer* or *Explorer* and select the *Preview* option from the dropdown list. The user can also magnify, rotate, delete pictures and view them as a slide show using this option.

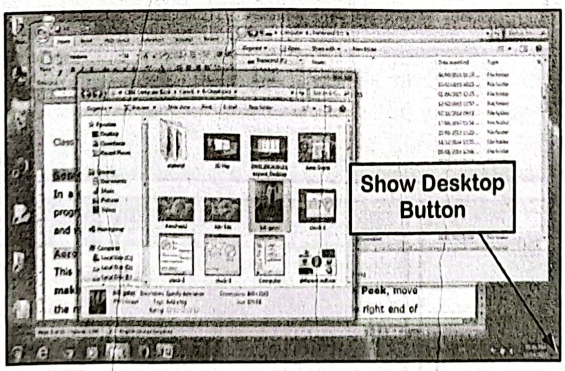
Some Helpful Features of Windows7

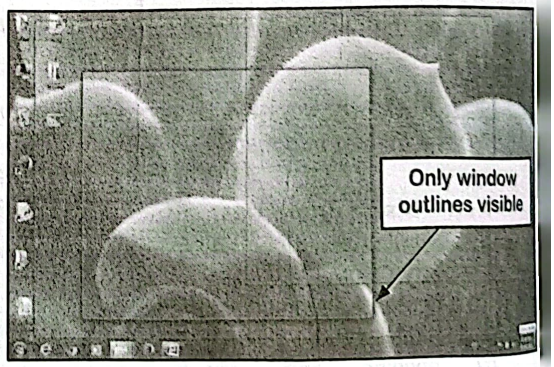
In a multitasking operating system like Windows, you can work with several programs at a time. **Windows7** uses the new **Windows Aero** feature to manage and work with different programs efficiently.

Aero Peek

Aero Peek

The Aero Peek option helps to have a quick preview of the Windows 7 Desktop by making all open windows transparent like glass. To use **Aero Peek**, move the mouse pointer over the **Show Desktop** button located at the right end of the **Taskbar**. All open windows will disappear and only the window outlines will be visible. Move the pointer away from the button and all the windows will reappear.



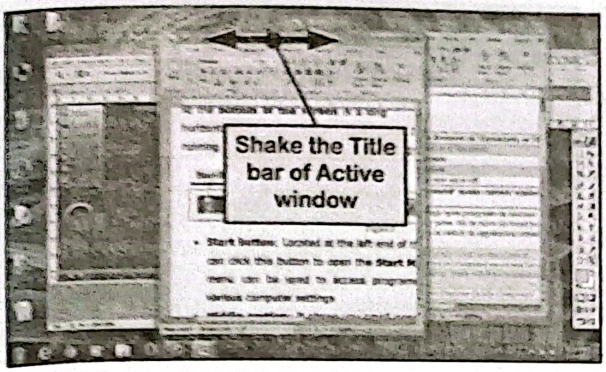


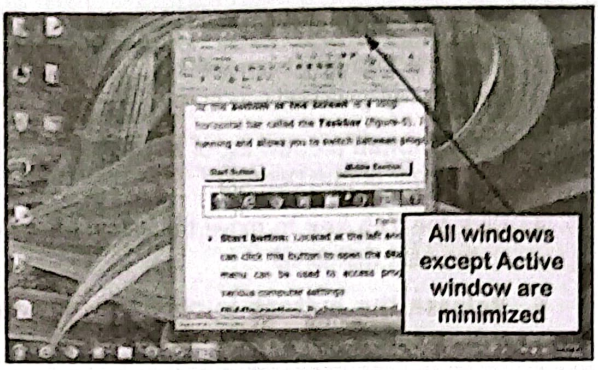
A **single click** on the button will remove all open windows from the desktop permanently and let you work on the desktop. A second click will restore all the windows again.

Aero Shake

Aero Shake

The Aero Shake option is used to minimise all open windows on the Desktop except the Active window. To use this option left click on the title bar of an open window and keeping the mouse button pressed, **quickly move it towards left and right** (i.e. shake it). All the other open windows will get minimised excepting the one which was shook. Shaking it again brings all the open windows back to the previous state.

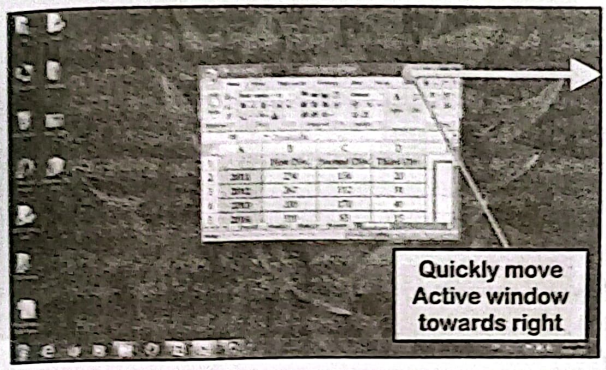


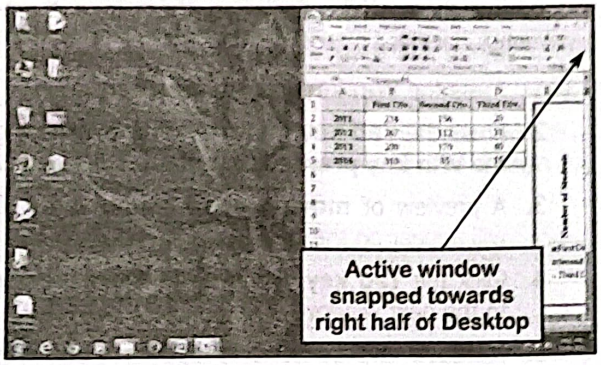


Aero Snap

Aero Snap

The Aero Snap option helps to maximise or restore a window without using the respective buttons, or align a window with the left or right half of the Desktop screen. The available options are:





Press the Win key + Right (or Left) Arrow key to snap Active window towards right (or left)

- Holding the title bar **quickly drag a window to the right** (or left) of the Desktop and release the mouse button, to fill the entire right (or left) half of the screen with the window (see figure above)
- To **compare** the contents of **two windows**, drag the windows using the Title bar to opposite sides of the screen. Each window will resize to fill half of the screen.
- Dragging a window to the **top edge** of the screen using the Title bar **maximizes the window**

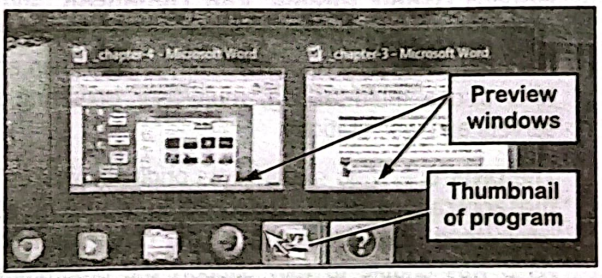
The original size of the windows can be restored by dragging the window towards the centre.

Live Taskbar Thumbnail

Live Taskbar Thumbnail

The **Windows7 Taskbar** helps you to get a **quick preview of open windows**. A thumbnail of the program icon is displayed on the taskbar for every open program.

If you point to such a thumbnail image, a **small preview window** of the item appears above the thumbnail. Pointing on the thumbnail preview window displays a full-screen preview.

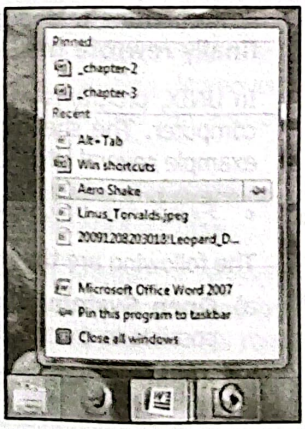


To open a program, click on a thumbnail or on the preview.

Jump Lists

Jump Lists

Jump Lists are **lists of recently opened programs**, folders, or websites that are displayed either in the **Start Menu** or on **Taskbar Thumbnails**. Using a **Jump List** one can open a recently used item by clicking on the entry in the list. One can also **pin favourite items** to a Jump List so that these can be quickly located and opened. A Jump List on a Taskbar thumbnail is shown on the right.

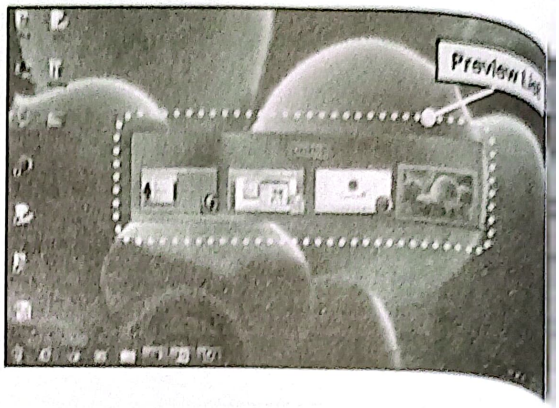


Using Window's Flip

Windows Flip

The Window's Flip feature is used to show a live thumbnail preview of each open window as a list on the desktop. To use this option:

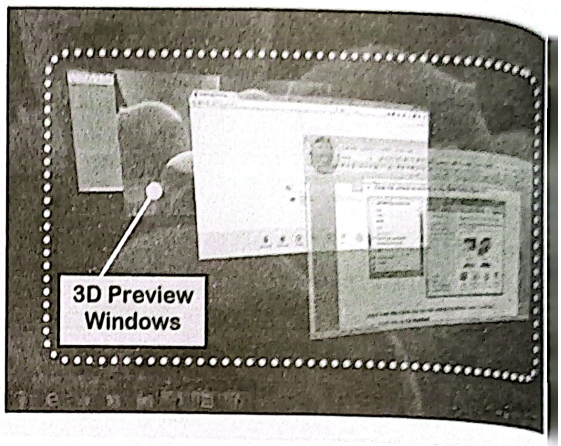
1. Press the **Alt key** from the keyboard
2. Press the **Tab key** from the keyboard keeping the **Alt key pressed**
3. A **Thumbnail** preview list of all open windows will appear on the Desktop
4. Click the **Tab key** to move the active preview to the next thumbnail
5. **Release the buttons** to open the currently selected preview window



Using Window's Flip 3D

The Windows Flip 3D feature shows live moving images of open windows, allowing you to switch between them while displaying them in a three-dimensional view.

1. Press the **Win key** from the keyboard
2. Press the **Tab key** from the keyboard keeping the **Win key pressed**
3. A preview of **moving open windows** in 3D will appear on the Desktop
4. Click the **Tab key** to move the active preview to the next window
5. **Release the buttons** to open the currently selected preview window



Press Alt+Shift+Tab to reverse the direction of the movement of the flip windows

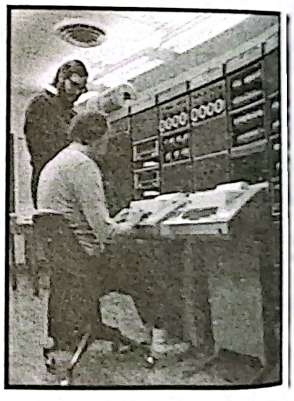
UNIX OS

6.5 UNIX Operating System

The next most important and widely used operating system is UNIX.

- **History and development of UNIX**

The UNIX operating system was originally developed by a group of employees at the AT&T Bell Labs during the end of the 1960's. The team included **Dennis Ritchie**, **Ken Thompson**, and others (see picture). The UNIX OS was designed to be portable in nature which made it possible to be used in a wide variety of machine families than any other operating system. It is also a multi-tasking, multi-user operating system and used in both servers and workstations.



The UNIX operating system was originally written in assembly language and the applications were developed in a mix of assembly language and an interpreted language called B. The programs were run on a PDP-7 system. As B was unable to fully support the programming needs, Dennis Ritchie developed the C language during 1971. **By 1973 Ritchie and Thompson finally rewrote the UNIX OS using the C language.**

In UNIX, programs can also be written in UNIX's command language to control the different actions of the computer. The shell provides the platform to write these command language programs and use them. For example several UNIX commands can be combined to form a shell script to do some specific job.

Features of UNIX

- **Features of UNIX**

The following are the **important features of the UNIX operating system**:

a) **Open System**: The user is able to write programs in UNIX to add to its capabilities. This makes it possible to extend the capabilities of UNIX and make it an open system.

b) **Reliable Security**: The use of password protected accounts in UNIX makes it highly secure.

c) **Powerful Kernel**: The powerful kernel helps in optimal resource management and acts as a guard to virus attack. This makes UNIX a secure and highly stable operating system.

d) **Multi User System**: UNIX was designed to be used in a multi-user environment. This helps in network implementation using UNIX.

e) **Multi Tasking System**: The multi tasking feature of UNIX makes it possible to simultaneously run multiple programs in a networked environment.

f) **Online Help**: The online help facility provided by UNIX makes it easy to get help while working.

• Working with UNIX


Working of UNIX

When you are connected to an UNIX system, as it is a multi-user operating system, it will first **present the user with the UNIX login message**. A multi-user OS allows each user to have a separate environment to work with and accordingly is given a **unique account**. The user can access the system through one of these accounts. Each **account is identified by a login user name** which has to be entered in response to the login prompt. In case the account is password protected, next UNIX will ask the user to type in the password.

Example: **login: pratik**
password: ******

Once the account name and the password are correctly entered, the user is finally logged into the UNIX system. After logging in, the user is **presented with the UNIX prompt** which is **generally the $ sign for a user account**, as shown below (it can be different for different shells or interfaces):

Example: **$ _**


Superuser

Unlike DOS, the UNIX system has two different types of prompts. One is for the user (generally the $ prompt). The other is for the system administrator, who is also known as the **root-user or superuser** and is **given the # prompt**. The superuser has access to all parts of the UNIX system and can access all files and commands in the system. The superuser also looks after system configuration and setting up, maintaining, and deleting of user accounts. Thus out of a total of 'n' nodes there are **n-1 user nodes and 1 super node**. For a newly installed UNIX systems the session begins by entering root as the login name for the root user.

Once logged in, UNIX automatically places the user in his **home directory** as determined by the system administrator while creating the user account. Whatever files and directories the user creates are stored in his/her home directory. Any modification that the user does to files in his home directory does not affect other users. Next the user has to type the commands in the UNIX prompt as required. Remember that unlike DOS, the **UNIX commands are case sensitive**.

Once the work is over, the user has to **log out of the system** by typing the following command:

Example: **$ exit**

• UNIX File System


UNIX File System

Files in UNIX are treated as a series of bytes. There is no distinction made between the types of files i.e. whether it is an ASCII file, binary file, or any other type of file. The meaning of the content of the file depends on the user of the file. File names in UNIX were initially up to 14 characters long but presently a **file name can be up to 255 characters long**.

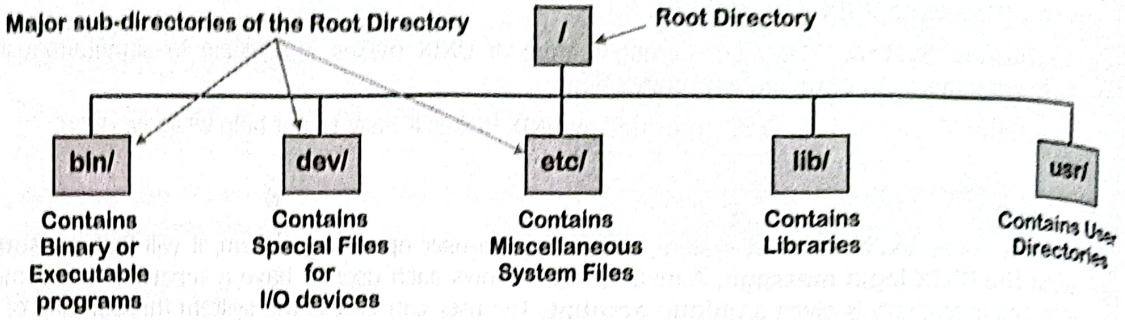
Unlike DOS, **filenames in UNIX do not have an extension**. If any extension is used, it will be taken as part of the file name. The user can specify file names as **program21.C** or **prog_15.C** to indicate files which are source codes of C programs, but to UNIX the **.C** portion will not have any special meaning. Moreover, such **extensions can be of any number of characters** and the number of such extensions can also be more than one. For example the file name **Project.final.C** is a valid file name in UNIX.

UNIX follows a hierarchical file system with directories and sub-directories. One can group files and directories and store them under other directories. However in UNIX the **directories can also be treated as files**.

UNIX file system supports a single inverted tree directory structure irrespective of the number of physical disks connected to the system. The most important directory in UNIX is the **root directory, indicated by the symbol '/'**. All other physical disks connected to the system are treated as subdirectories of this root directory. The same symbol '/' is also used to separate directories and sub-directories (note that DOS uses '\').

Unlike DOS, UNIX has **a standard directory structure.** Some of the standard directories under the 'root' directory provided by the UNIX directory structure are listed below (note that the '/' after the sub-directory name indicates that it is a directory and not a file).

Standard directory structure in UNIX



In a multi-user operating system separate rights need to be given to different users, based on who should be allowed to read files, write into files or run certain programs. This is done to ensure that the files remain secure. UNIX handles this problem by giving **different file permissions to the different users.**

3 types of File Permissions

There are **three types of file permission:**

- **Read Permission:** Read permission means a user can read the file only and cannot modify it.
- **Write Permission:** Write permission allows the user to modify a file in addition to reading it.
- **Execute Permission:** Execute permission means one can run the file as a program.

There are also **three levels of permissions:**

- **Owner Level Permission:** Owner permissions determine the activities permitted by the owner or the creator of the file.
- **Group Level Permission:** Group permission determines the activities permitted by the members of a user group. The system administrator may assign a user account in one or more user groups such that only members in a user group can have access to certain files.
- **Others/World Level Permission:** Others/World permission determines the activities permitted by users who are neither the file's owner nor are in the file's group.

When a file is listed, the first column of the listing has a fixed set of **10 characters** showing the file type and the type of permission granted. The meaning of each character is given in the table below:

Position	Description
1	This character indicates the **type of the file.** The most common types are '–' for a regular file, '**d**' for a directory, and '**l**' for a symbolic link. It can contain other characters like '**c**' which indicates a character device like a terminal, '**b**' which indicates a block device like a disk, etc.
2 to 4	Permission set for the directory or **file's owner.**
5 to 7	Permission set for the directory or **file's group.**
8 to 10	Permission set for **all users** who are neither the file's owner nor a member of the file's group.

3 levels r,w,x for each permission type

Each **permission set consists of three characters** the meaning of which is given below:

- **r** : Indicating **r**ead permission for that set. If it is 'r', then the file may be read by the user having this permission. If it is '–', then the file may not be read by the user.
- **w** : Indicating **w**rite permission for that set. If it is 'w', then the file may be written to or deleted by the user having this permission. If it is '–', then the file may not be written to or deleted by the user.
- **x** : Indicating **e**xecute permission for that set. This value can be 'x' when the file can be executed, it can be 's' for execution with some additional privileges or may be '–' when it may not be executed.

The following examples will illustrate some of the types of permissions possible.

Example: d r w x r w x r w x

Dir. Dir. owner User group All users

In this example, the type of the entry is a **d**irectory.
The directory owner has permission to **r**ead, **w**rite and **e**xecute.
The directory user group has permission to **r**ead, **w**rite and **e**xecute.
Also other users have permission to **r**ead, **w**rite and **e**xecute.

Example: d r w x r – – r – –
Dir. owner | Dir. group | User | All users

In this example, the type of the entry is a **directory**.
The directory owner has permission to **read**, **write** and **execute**.
The directory user group has permission to only **read** the file.
Also other users have permission to only **read** the directory.

Example: – r w x r w x r – –
File owner | File group | User | All users

In this example, the type of the entry is a **file**.
The file owner has permission to **read**, **write** and **execute**.
The file user group has permission to **read**, **write** and **execute**.
But other users have permission to only **read** the file.

Example: – r w x – – – – – –
File owner | File group | User | All users

In this example, the type of the entry is a **file**.
Only the file owner has permission to **read**, **write** and **execute**.
The file user group has no permission to **read**, **write**, or **execute**.
Other users also have no permission to **read**, **write**, or **execute**.

• Booting of UNIX

Booting of UNIX

The exact booting sequence in UNIX depends on the type of distribution and the system on which it is running. Given here is a generalised description of the booting process in UNIX based on a particular distribution.

When you start the computer running an UNIX operating system, the **first sector of the boot disk is read** and loaded into the memory. This sector contains a small program which executes and in turn loads a **special program called the boot** from the boot device (which is usually the hard disk). The boot program reads the root directory of the boot device to understand the file system and **loads the kernel** which in turn takes over and starts execution of the main portion of the operating system. Next the system starts auto-configuration to **locate the I/O device drivers**.

After this hardware configuration phase, three different processes start. The first process i.e. **process-0** launches a program called **init** which then launches the other processes. Therefore **init** is a parent to all the other processes. It is also responsible for setting up the real time clock, and loading the root file system. The next process i.e. **process-1** then checks if it is a single user or multi-user system and takes appropriate actions accordingly. It is also responsible for starting **process-2**. Then finally the **login prompt is displayed** by the process to read the user name and password. If login is successful, the user's shell is loaded, which finally waits for any further command from the user.

• UNIX Shell Commands

Let us now discuss the various commands in UNIX called the **UNIX Shell commands** that can be used to do the various operations discussed earlier for the other operating systems.

▪ ls

ls command

The **ls** command is used to **list the contents of a specified directory**. If no directory is specified, the contents of the current directory are displayed. It is similar to the **DIR** command in DOS. File listing can be viewed in various forms based on the use of **several switches**, some of which are described below:

- **–a** Lists all contents including hidden files
- **–l** (el) Lists the contents of a directory in long form
- **–d** Lists only the name of the directory and not its contents
- **–t** Lists the contents in order of time saved, beginning with the most recent one
- **–s** Lists the contents in order of size
- **–1** Lists the contents one entry to a line

Example: $ ls

Output: prog01.C prog02.C yogesh

The above ls command simply displays the contents of the current directory and displays the names of 3 files, viz. **prog01.C**, **prog02.C**, and **yogesh**. (**Do not** type the $ sign, as it represents the prompt, and will be automatically displayed on the screen).

Example: $ ls -l

Output:

```
-rw-r--r--   1   user1   group1   62    Mar 12   11:09   prog01.C
-rw-r--r--   1   user1   group1   55    Apr 11   15:21   prog02.C
drw-------   1   user1   group1   265   Sep 2    08:17   yogesh
```

The above **ls** command displays the contents of the current directory in **long format** using the **-l** (el and not 1) switch and displays the names of the 3 files along with the file details. Each **column in the listing displays a certain property** of the file. The **last column displays the name of the file**. The next two columns display the time and date on which the file was created. The next column displays the number of bytes of that file. The next two columns display the group and user names. In this case these are **group1** and **user1**. The next column displays the link counter, which displays how many files are symbolically linked to the given file. The final column (i.e. the actual first column in this case) displays the **10 character file type** and file permission information as discussed earlier. For example the file **prog01.C** has read/write permission for the user and only read permission for the group and for all other users. The example also shows that the entry **yogesh** is a directory and not a file (not evident from the short form of listing in the previous example).

Example: $ ls /usr/yogesh

Output: **pic1.jpeg birthday.jpeg biodata.text Darjeeling.bmp Budget_June09.dbf**

The above **ls** command displays the contents of the sub-directory **yogesh** which in turn is located under the **usr** directory under the root '**/**' (in this case 5 entries displayed). Note that the '**/**' symbol is used repeatedly to separate the sub-directories and to **indicate the full path of the sub-directory**.

The same listing can be displayed by using the relative path concept. If the user is within the **usr** sub-directory, then the listing command can be alternatively written as:

Example: $ ls ./yogesh

Output: **pic1.jpeg birthday.jpeg biodata.text Darjeeling.bmp Budget_June09.dbf**

The '**.**' in the above **ls** command indicates the path of the current directory so that the path typed above displays the relative path of the **yogesh** sub-directory.



mkdir command

- **mkdir**

The **mkdir** command in UNIX is used to **create a directory**. While creating a directory, the **-p switch** can be used to create the parent directories also, if these are not already created. It is similar to the DOS **MKDIR** or **MD** command. Some of the uses of this command are stated below:

Example: $ mkdir yogesh

The above command creates the directory called **yogesh** under the current directory.

The command can also be used to create directories within directories other than the current directory. The next example creates such a directory called **pic** under the **yogesh** directory located under the **usr** directory.

Example: $ mkdir /usr/yogesh/pic

Multiple sub-directories can be created under current directory by writing the sub-directory names separated by spaces. The next example creates sub-directories **letters, pictures** and **documents** under current directory:

Example: $ mkdir letters pictures documents

When the command is used with the **-p** switch, then both the sub-directory and the parent directory can be created. The following example will create both the sub-directory programs and the parent directory called **sauro** under the current directory as shown below:

Example: $ mkdir -p /sauro/programs



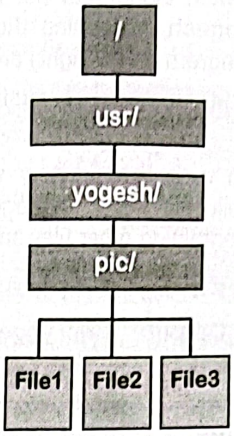
cd command

- **cd**

The **cd** command is used to **change the current directory** of the shell. This current directory is also used by other programs launched from the shell. It is similar to the DOS **CD** command.

We have seen that the root directory is represented by '/' symbol. Therefore the following example can be used to move to the root directory:

Example: $ cd /



The following example can be used to move to the parent directory from the current directory. The two dots '..' following the **cd** command is used to indicate the parent directory (this is similar to the **CD** command of DOS).

Example: $ cd ..

To move to any other directory from the current directory one can use the path of the directory along with the **cd** command. In the next example, the user can move to the directory **/usr/yogesh/pic** (see diagram on the right) from the current directory by using the **cd** command as:

Example: $ cd /usr/yogesh/pic

- **rmdir**



rmdir command

The **rmdir** command is used to **remove an empty directory** i.e. a directory containing no files or sub-directories. In case a non-empty directory with files or sub-directories is deleted using the **rmdir** command, then UNIX will show an error message and will not delete the same. To remove a directory using this command, first remove all files within that directory using the **rm** command and then remove all other sub-directories using the **rmdir** command. The user must also be in another directory to carry out the command.

The following example can be used to remove the **pic** directory only after removing all files and sub-directories in it.

Example: $ rmdir pic

To remove another directory from the current directory one can use the path of the directory along with the **rmdir** command. In the next example, the user can remove the directory **/usr/yogesh/pic** from the current directory by using the **rmdir** command. However the **pic** directory should be empty. If **pic** contains any files then those need to be removed first.

Example: $ rmdir /usr/yogesh/pic

- **rm**



rm command

The **rm** command is used to **remove both files and directories**. Care should be taken while using the command as files can get deleted accidentally. Two switches used commonly with this command are:

–r Removes all files and directories in a specified location

–i Removes files only after verification from the user

Example: $ rm birthday.photo

The above example will remove the file **birthday.photo** from the current directory. It will however not ask you for any confirmation before deleting the file.

Multiple files can also be deleted using this command by writing the file names to be deleted separated by blank spaces as shown below:

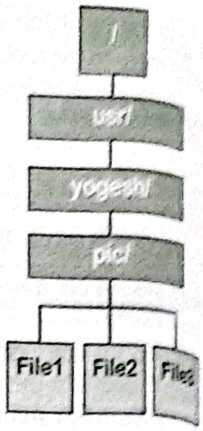
Example: $ rm prog01.C prog02.C prog07.C

To prevent any accidental removal of a file use the **–i** switch as shown below. UNIX will ask for verification before deleting the file. Type the character **y** at the prompt to confirm the deletion.

Example: $ rm -i prog01.C prog02.C

Output:
```
prog01.C ? y
prog02.C ? y
```

However in case a directory contains files and sub-directories, then **to remove such a directory along with the sub-directories and files**, one has to use the –r switch along with the **rm** command. In the example shown below, the directory **yogesh**, containing the sub-directory **pic** and all files within **pic** (see directory diagram on the right) are removed using the command along with **-r**.



Example: $ rm -r /yogesh

In case the directory **yogesh** is removed without using the –r option, then the following error message will be displayed to indicate that **yogesh** is a directory containing other files and directories:

Example: $ rm /yogesh

Output: rm: yogesh directory



cp command

cp

The **cp** command is **used to copy files**. The command can be used to copy files to keep a backup of a file or can be used as a template for creating another file. Some switches used with this command are:

- –r Copies directories (recursively)
- –i Prompts before overwriting a file while copying
- –v Shows the filenames as these are being copied

Example: $ cp biodata.docu old_biodata.docu

The above **copy** command will simply make a copy of **biodata.docu** with the new name **old_biodata.docu**

Example: $ cp -i rik.report.bak /tmp/goirik

Source file to copy **Destination directory**

The above **copy** command will make a copy of the file **rik.report.bak** present in the current directory to the target directory **/tmp/goirik** with the same file name. Moreover using –i UNIX will prompt you before copying, in case it has to overwrite a file. Press **y** to carry out the command as shown below.

Output: cp: overwrite rik.report.bak ? y

Example: $ cp -i rik.report /tmp/goirik/rik.report2009

The above copy command will make a copy of the file **rik.report** as in the previous example, but in this case the file will be copied with a different file name as **rik.report2009.**

Multiple files with different names can also be copied from the current directory to another directory using this command. In the example shown below, three files are copied from the current directory to the sub-directory **goirik** under the **tmp** directory.

Example: $ cp -i rik.bio rik.report computer.notes /tmp/goirik

The next example is used to make a **copy of the directory** pictures (under the current directory) and all its contents to the **/tmp/old_pics** directory using the –r switch.

Example: $ cp -ir pictures/ /tmp/old_pics

Source directory to copy **Destination directory**



mv command

- **mv**

The **mv** command is used to **move or rename files**. It can be used to move files from one directory to another. Unlike a **copy** command, during a **move** operation no copy of the file is kept in the source directory and the file is totally moved to the target directory.

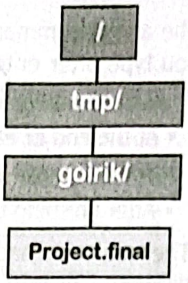
While moving a file, the user can also rename the file using this command. The **rename** command can also be used to rename a file in the same directory. One can also **move directories** and all its contents using the same **mv** command. The switches used commonly with this command are:

–f Moves files without checking for confirmation in case of an overwrite
–i Prompts before overwriting a file while moving

The following example can be used to rename the file **project.draft** in the current directory to **project.final**. No copy of the file is made, but the filename is changed.

Example: `$ mv project.draft project.final`

In the next example the file **project.draft** is moved from the current directory to the sub-directory **/tmp/goirik**



Example: `$ mv project.draft /tmp/goirik`

In the next example the file **project.draft** is moved from current directory to sub-directory **/tmp/goirik**, but simultaneously the name of the file is also changed to **project.final**

Example: `$ mv project.draft /tmp/goirik/project.final`

Just like the **copy** command, while moving a file to the target directory if there is an existing file with the same name then UNIX will simply overwrite the existing file without asking for any permission. To avoid this, the –i option is used, whereby UNIX will ask for a permission to overwrite an existing file as shown below.

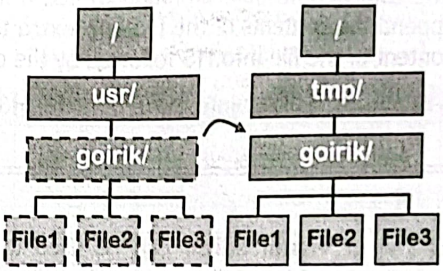
Example: `$ mv -i project.draft /tmp/goirik/project.final`

Output: `mv: overwrite project.final ? y`

The next example shows how to move a directory along with all its contents from one location to another. It moves the directory **goirik** along with all its files from **usr** directory to **tmp** directory.

Example: `$ mv /usr/goirik /tmp`

Source directory to move Directory into which to move



- **cat**


cat command

The **cat** command (short for concatenate) is used to **display the contents of a file** containing ASCII text onto the screen, without the need to open a text editor. It can also display multiple files concatenated together. Some useful switches used with this command are:

–n Used for line numbering
–s Used to squeeze sequences of blank lines into one
–v Used to show non-printing characters

The following example is used to print on screen the contents of the file **pratik.cv** (provided the file had been created earlier by a text editor).

Example: `$ cat pratik.cv`

Output:
```
Name: Pratik Banerjee
Class: 11
Section: E
Fourth Subject: Computer Science
```

The **cat** command can also be used to make a copy of a file by directing the output of the **cat** command to another file instead of displaying it on the screen. In this respect it works similar to the **cp** command. The following example creates a backup file using the **cat** command.

Example: `$ cat pratik.cv > pratik.cv.bak`

No output will be seen on the screen, but the contents of the file **pratik.cv** will get copied to the new file **pratik.cv.bak**. In case an existing file is there with the same name, then it will get overwritten by this command without any prompt.

The **cat** command can also be used to create a file containing simple ASCII text instead of using a proper text editor. For example to create the file **pratik.cv**, we can use the **cat** command in the following manner.

Example: **$ cat > pratik.cv**

Output: *Name: Pratik Banerjee* ↵
Class: 11 ↵
Section: E ↵
Fourth Subject: Computer Science ↵
Ctrl+D

The above command creates a file called **pratik.cv** and sends the output from the keyboard (i.e. whatever you type after entering the command) one line at a time to that file. The following rules should be kept in mind while entering the data:

- At the end of each line press the Enter key (↵)
- You can move around the current line using the backspace key. Moving in-between lines is not allowed
- After finishing the typing, press **Ctrl+D** to save the file and get back to the prompt

The **cat** command can also be used to combine two or more files into single file. Suppose there are two different files **info.11** and **info.12** containing the ID, name, class, and section information of students of class 11 and 12 respectively. These two files can be combined to form a single file called **info.HS** using the **cat** command as shown below:

Example: **$ cat info.11 info.12 > info.HS**

One can use the **cat** command to add a file at the end of another existing file. The following example will append the contents of the file **info.extra** to the end of the file **info.HS**. The resulting file will have the initial content of the file **info.HS** followed by the contents of the file **info.extra**.

Example: **$ cat info.extra >> info.HS**


vi editor

- **vi**

The **vi** or **visual editor** can be used to **create, load and edit ASCII files** in UNIX. Like most word processors, **vi** is a full screen editor allowing an ASCII file to be loaded and viewed one screen at a time. However **vi** is extremely fast while scrolling through large documents. **Vi** however does not support document formatting (bold, italic, underline etc.) and spell checking, which are common features in a word processor.

To start **vi** and edit the new file puzzle, type the following command. The home directory is the current directory in this case.

Example: **$ vi puzzle**

```
~
~
~
~
~
~
~
~
~
"puzzle" [New File]
```

The screen shown to the right opens. The screen displays a series of tildes (~) in the column 1 of the text editor.

Vi operates in two modes – namely text **insert mode** and **command mode**. The normal mode for **vi** operation is the command mode. The text insert mode is entered from the command mode when the user wants to insert or replace text. Press the **Esc** key to come out of the text insert mode to the command mode.

The tilde in the first column of every line indicates that no text is present in that line. The tilde disappears when some character is typed in each line. To insert text into the **vi** editor first press the **i** key and then start entering the text. Press the Enter (↵) key to move to the next line.

Question: Can you name three consecutive days without using the words Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, or Sunday? ↵

Answer: Yesterday, Today, and Tomorrow ↵

Accidental errors can occur while entering text. Later these can be corrected using proper commands. However while entering a line of text the user can use the Backspace key to move back and correct an error. But once you press the Enter key after finishing writing a line, you cannot use the Backspace key to move back. After finishing writing, press the Esc key to get back to the command mode. The vi screen now looks like the one shown on the right.

```
Question: Can you name three
consecutive days without using
the words Monday, Tuesday,
Wednesday, Thursday, Friday,
Saturday, or Sunday?

Answer: Yesterday, Today, and
Tomorrow
~
~
"puzzle" [New File]
```

To make the entered text permanent it needs to be saved. To save the file type :w and press Enter. Finally to quit the vi editor type :q and press Enter.

To edit the text before you save it, you can use the following keys/commands. Get to the command mode to move around the text and then use the text insert mode to edit the text.



Some vi editor commands

Cursor Keys and Commands			
Key	**Cursor Movement**	**Key**	**Cursor Movement**
←, ↑, →, ↓	Left, Up, Right, Down	$	To the end of the current line
0 (zero)	To beginning of current line	G	To the end of the file

Some Edit Commands			
Command	**vi Function**	**Command**	**vi Function**
i	Enter text entry mode and insert text at the current cursor position	x	Delete the character at the cursor position. It may be preceded by a number to delete multiple characters
I	Move to the first character in the current line and enter text entry mode	~	Change the case of the character at the cursor position
a	Append text after the current cursor position	Esc	To leave the text entry mode
A	Append text at the end of the current line	:wq!	Save the changes to the current file and exit the editor
r	Replace a single character	:q!	Exit without saving the changes

▪ more



more command

The **more** command is used **to paginate the output**. The problem with the **cat** command is that if a file is too long, then it falls beyond the top of the screen. The job of more is to stop and wait when it fills the screen. The command will act like **cat** if the file is short. The Enter key can be used to read the next line and the spacebar can be used to move to the next page. To quit the command press **q**.

The following command is used to view the contents of the file **soumya.hobby** one page at a time.

Example: $ more soumya.hobby

▪ chmod



chmod command

The **chmod** command is used **to change file permissions**. The command uses a numeric scheme to specify various modes of permission. A mode is **represented by an octal number** as per the table shown to the right.

For example to grant **r**ead, **w**rite, **e**xecute permissions to the owner and only **r**ead permission to the group and all users, the required mode value is calculated by **adding the numeric values** that correspond to each type of permission from the table shown to the right (the grey shaded cells) as

Numerical	Meaning
400	Owner has read permission
200	Owner has write permission
100	Owner has execute permission
040	Group has read permission
020	Group has write permission
010	Group has execute permission
004	All users have read permission
002	All users have write permission
001	All users have execute permission

(400+200+100+040+004)=744. The command line to apply this permission on the file **biodata** will be:

Example: $ chmod 744 biodata



who command

- **who**

The who command is used to **display who else are logged onto the system** at a given time.

Example: $ who

Output:	**pratik**	**term/08**	**Jul**	**20**	**11:45**
	goirik	**term/05**	**Jul**	**20**	**12:15**
	sanjit	**term/02**	**Jul**	**20**	**09:35**

The output gives the username, terminal ID no. and when that user had logged into the system.



cal command

- **cal**

The cal command is used to **display a one month calender** on the screen.

Example: $ cal



Wild card characters

- **Wild Card characters in UNIX**

Using wild card characters it becomes easy to **deal with a set of files and directories with a single command**. Wildcards allow the user to specify similar files without having to type multiple names. It also allows the user to search for files even if the exact name is not known. There are three types of wildcards used in UNIX. These are '*', '?', and a pair of square brackets i.e. '[...]'. The following example will display names of all files that start with the word project.

Example: $ ls project*

Output: project project.draft project.final project.hs

In the previous example we have seen that the '*' **wildcard is used to match any number of characters** in the filename. Moreover it can be used anywhere in the string as shown below.

The following example will display names of all files that contain the string **prog**. The '*' in front of the string and also at the end indicate any number of characters in front or at the end of the string.

Example: $ ls *prog*

Output: prog21.C final.prog program49.CPP backup.prog23.C

Unlike the '*' wildcard, the '?' **wildcard is used to match a single character**. The following example lists all files that start with the string **prog** and have a **single extra character at the end**.

Example: $ ls prog?

Output: prog1 prog2 prog6 prog9

The following example similarly will list all file names that start with the string **prog** and have only **two extra characters at the end**.

Example: $ ls prog??

Output: prog10 prog12 prog32 prog41

The '*' and '?' characters **can also be combined** to display specific names. The following example combines these to display the names of all files that contain the string layout. There can be any number of characters before the string layout as indicated by the '*' wildcard, but there can be only one character after the string layout as indicated by the '?' wildcard placed after it.

Example: $ ls *layout?

Output: draft.layout1 draft.layout2 final.layout1 submit.layout1

Apart from the above two wildcards, the third one used in UNIX is the **[...]** wildcard **to match specific characters.** This option becomes helpful when the user is looking for files without being sure about the case of the characters. As UNIX is case sensitive, this can be useful.

The next example looks for files with starting character either 'S' or 's' followed by **anjit.class11.**

Example: **$ ls [Ss]anjit.class11**

Output: **Sanjit.class11**

A range of characters can also be specified using the '–' sign. The following example can look for files which end with the characters **a, b, c,** and **d.**

Example: **$ ls project.[a-d]**

Output: **project.a project.b project.c project.d**

Though we have done the discussion with only the **ls** command, **the above wildcards can also be used with the other commands** like **cp, mv, rm** etc.

- **Piping operation in UNIX**


Piping operation

The idea of pipes plays an important role in UNIX programming. A pipe is set up to indicate that the **output of one command will be used as the input for another command.** Unlike redirection (see examples of the **cat** command) which sends the output to another file, a pipe sends the output to another command. Thus a pipe can be thought of as a temporary file that holds the output of the first command for the second command. There are no limitations to the number of pipes in a command line. The character '|' (placed with the '\' key) is used to indicate a piping operation.

The following command is used to view the listing of the directory **/usr/yogesh** page by page. The output of the **ls** command (with the **–l** option to create a long listing) is piped into the more command, which displays the listing one screen at a time. The use is similar to the use of the DOS DIR command along with **/p** switch.

Example: **$ ls -l /usr/yogesh | more**

- UNIX Shell


The UNIX Shell

After logging into UNIX, the user is put into his/her login shell. **A shell is basically a program like any other UNIX programs.** However, it is a special program in the sense that it serves as the interface between the user and the UNIX system and **translates the instructions given by the user in a way that the UNIX system can understand.** It tells the UNIX system, where to find specific files, where the home directory is, and in general how to deal with the user's presence.

There are **several shells available in UNIX.** The user usually sticks to the shell that the system administrator has chosen for the user. Some of the popular shells are described below.

- **The Bourne Shell:** This shell was shipped with the original System-V release of UNIX in 1979 and has remained almost unchanged since then. It was named after its developer Stephen Bourne. The UNIX name for this shell is sh.
- **The Korn Shell:** It was developed by David Korn by adding several useful features over the functionalities of the Bourne shell. Therefore scripts developed for the Bourne shell can run without any problem in the Korn shell. This shell is represented by the UNIX name ksh.
- **The C Shell:** This shell was developed by Bill Joy while working on the release of Berkeley UNIX (also called BSD). The C shell includes features like job history, aliases and some other features. It is also structured in line with the C programming language. The C shell is therefore different from both the Bourne and Korn shells. It is represented by the UNIX name csh.
- **The Job Shell:** It is an extension of the Bourne shell and has features developed for handling multiple jobs. It is represented by the UNIX name jsh.

The following command can be used to find the shell where the user is logged into. Usually shell names end with **sh.** Thus if you are in the **Korn** shell, then the output of the command will be:

Example: **$ echo $SHELL**

Output: **ksh**

Comparison DOS & UNIX

Comparison between DOS and UNIX Commands

DOS Command	UNIX Command	Use
ATTRIB	chmod	Used to change file permission
CD/CHDIR	cd	Used to change the directory
CLS	clear	Used to clear the screen
COPY	cp	Used to copy the contents of a file
COPY CON	cat	Used to create a file
DEL/ERASE	rm	Used to delete a file
DELTREE	rmdir	Used to delete the contents of a directory
DIR	ls	Used to display a directory listing
EDIT	vi	Used to create and edit a text file
MD/MKDIR	mkdir	Used to make a directory
MORE	more	Used to display one screen of data at a time
RD/RMDIR	rmdir	Used to delete a directory
REN/RENAME	mv	Used to rename a file or a directory
TYPE	cat	Used to display the contents of a file

The LINUX OS

6.6 The Linux Operating System

Linux is an operating system that **belongs to the UNIX family of operating systems**. Before the advent of Linux, the powerful features of UNIX were not available for home computers.

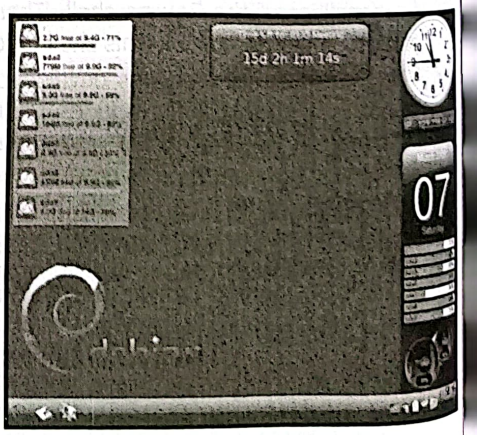
- History and development of Linux



Richard Stallman & Linus Torvalds

The name Linux comes from the **Linux kernel** that was originally **written by Linus Torvalds**, then a student at the university of Helsinki. The first Linux kernel released to the public was Version 0.01 on May 14, 1991. To make a complete operating system the other components of an operating system, like libraries, compilers, text editors, a UNIX shell, and a windowing system were also incorporated into the system. These components were added from the **GNU Project** that was started by **Richard Stallman** in 1984. GNU programs are all released under a special licence called the **GNU Public Licence** which describes the principles of free, open-source software. The licence was written in 1989 after Stallman started the Free Software Foundation. Therefore the aim of the GNU Project was to create complete UNIX compatible software, that **contained only free software.**

By the term 'free', it means two things. Firstly, the **user does not have to pay for the software**, and secondly **the user can have full access to the source code** of Linux. The user can copy it, modify it, and again redistribute it to another person if he wants. The only condition is that the source must be freely available to all.



Linux vendors and communities combine and distribute the kernel, GNU components, and non-GNU components, with additional package management software in the form of **Linux distributions** commonly called 'distro'. Although by the terms of the licence no one can charge for Linux, but the **user of the distribution has to pay for the distribution.** This distribution can be in a CD/DVD ROM, in a box with a manual or can be via the Internet. Accordingly the distribution charges can vary. Some vendors, like **Debian**, develop and fund their distributions on a volunteer basis. While others, like Red Hat with Fedora, maintain a community version of their commercial distributions. In many cities and regions local associations known as **Linux Users Groups** (LUGs) seek to promote their preferred distribution.

The **Linux logo Tux** (the penguin) is shown on the right. Many types of applications available for Microsoft Windows are also available for Linux. Commonly, either a free software application will exist which does the functions of an application found on another operating system, or that application will work on Linux.



- **Structure and Working of Linux**

Working of Linux

Linux uses a **monolithic kernel**, the Linux kernel, which handles process control, networking, and peripheral and file system access. As the Linux kernel is under constant development, a numbering pattern is followed to identify the nature of the kernels. Kernels with an odd numbered minor version, like 1.1, 1.3, 2.1, are development kernels; whereas even numbered minor version ones are stable production kernels. The device drivers are integrated directly with the kernel. Users can operate a Linux-based system **either through a command line interface or through a graphical user interface**. The graphical user interface used by most Linux systems is based on the X-Window System. For desktop systems usually a graphical user interface is used and **KDE, GNOME, and Ubuntu are some of the most popular graphical user interfaces**.

After installing Linux when it is started, the user will see the login prompt. Just like UNIX, for the first time login the only user on the system will be the **super-user** known as root. To log in, the user has to **enter his username at the login prompt and the password at the password prompt**. After a successful login, the user will be placed at the Linux prompt. After logging in for the first time, a new user needs to be added.

The following command is to be used to add a user:

Example: useradd -m sauro

Next, the password for this user is set by using the following command:

Example: passwd sauro

Finally press **Ctrl+D** to logout from the system.

Similar to other UNIX systems, Linux disks are organised into a **logical tree structure of directories**. Each directory can in turn hold other sub-directories or files (both program and data). As usual, the **base of the tree is called the root** and is represented by the '/' sign.

Again, just like UNIX, some root level directories found in Linux are, **bin, boot, dev, etc, home, lib, root, tmp, usr, sbin** etc. each having its own specific content similar to the UNIX file system. Therefore the following path shows the location of the file **biodata.dat** within the **sauro** sub-directory which in turn is located under the **home** root directory.

Example: /home/sauro/biodata.dat

A Linux filename can be any length long but are case sensitive. In case they contain spaces, when used with a command they may need to be enclosed within double quotes for a command to know that it is a single filename. Therefore **"project", "Project"** and **"PROJECT"** are all different names and can co-exist.

- **Command Line Interface commands in Linux**

CLI commands in Linux

Just like UNIX, after the user logs in into his new username, he will be placed at the Linux shell prompt. The prompt will be displayed with the $ sign, but prefixed with the username, the name of the Linux machine and the current directory name.

Two types of commands can be entered at the shell prompt. These are **Shell Commands** and **Programs**. The shell commands instruct the shell to do something and are used for job control, and control of how the shell operates and the user environment. Whereas typing a program name causes the shell to go and find that program on the hard disk and run it.

The basic file handling commands in Linux are the same as that used in UNIX and hence are not discussed elaborately here. **The description for some of those commands is given below:**

Command	Use
date	Used to display the current system date and time
ls	Used to list directory contents
cp	Used to copy files
cat	Used to display the contents of files

mv	Used to rename files
rm	Used to delete files permanently
mkdir	Used to create a directory
rmdir	Used to remove a directory
cd	Used to change the current working directory
more	Used to display the contents of a file one page at a time
find	Used to search for a file within a given directory and its sub-directories
vi	Used to start the standard text editor
chmod	Used to change file permission



The Fact File

- **The hardware part of a computer on its own is useless and cannot do any useful work. The hardware needs to be driven by means of software programs, which specify the tasks to be done by the computer**
- System software are used to run the computer and manage the different resources of a computer
- **The Operating System (OS) is the most significant system software and acts as a link between the hardware and the user**
- The OS sets the environment in which the programs work and the user interacts with the computer
- **Library Programs are used to do certain utility jobs that include finding files, compressing files, disk defragmenting etc.**
- Editor is a text editing software that is used to create, open, or edit a text file
- **An Assembler is used to translates an Assembly Language Program to Machine Code**
- An Interpreter is used to translate a program written in a high level language to machine code and simultaneously execute the converted code
- **A compiler is a program used to convert a program written in a high level language to machine language. But a compiler only creates an object program, but does not execute the program**
- The output of the compiler is the machine language code also called the object program
- **A Linker is a system software that is used to combine or link two or more object programs to form a Load module**
- Application and General Purpose Software are user developed programs that are made to carry out specific jobs like stock management, school management, hospital management, payroll etc
- **When a program is executed, it is broken down into a number of smaller units called processes or jobs. Each process acts as a unit of work for the processor and shares the computer resources like the CPU, memory space, files and input/output devices**
- Any new process entering a system must initially go to the Ready state. In this state it waits in a queue along with other processes for the CPU time to be assigned to it
- **In the Running state the process has control over the CPU and uses it to get executed**
- In the event of an input/output operation, or an interrupt, the Running state of a process is changed to Blocked state
- **The Memory Management Module of the OS is responsible for the allocation and de-allocation of memory space to various programs**
- The Device Management Module of an OS keeps track of the input/output requests from various processes, issues commands to the input/output devices and ensures correct data transfer to and from the those devices
- **The File Management Module of the OS deals with creating, naming, storing, retrieving, organising and the security of files**
- The Security Management Module of the computer is responsible for protecting the computer system from misuse, errors in programs and ensures a smooth run
- **The Command Interpretation Module of an operating system takes care of interpreting the user commands and directing the system resources to handle the requests**
- Mainframe Operating Systems serve mainframe computers
- **Batch Processing Operating Systems are used for processing a set of jobs without any human intervention like payroll file processing, weather forecasting, statistical analysis etc.**
- **To separate one job from another for a batch of jobs, special languages called job control languages (JCL) containing control statements are used**
- As a single job is processed at any given time by a batch processing OS, there is no competition for input/output devices. Therefore memory management, file management, and I/O management are very simple in such a system

- **Multiprogramming Operating Systems use interleaved execution of two or more different independent jobs or programs by the same computer**
- A multitasking operating system can handle multiple tasks together by applying multiprogramming techniques
- **Multiuser Operating System allows multiple users to access a computer through more than one terminal. Examples include the Railway booking systems**
- Multiprocessing Operating Systems have multiple independent processors that work in parallel
- **Time Sharing Operating Systems allow various users to share the CPU, memory and other resources of the computer system under its supervision**
- Real Time Operating Systems are used in places where the response time required for data processing is critical. Examples of such processes are air traffic control, nuclear power plant monitoring
- **Network Operating Systems are used in computers connected in a computer network, like a LAN**
- Distributed Operating Systems have several computers that are connected together and process a given job by sharing the job load between them
- **Personal Computer Operating Systems are used in PCs and are usually meant for home use**
- The Kernel is the innermost layer and is the central controlling part of the operating system. It always resides in the main memory and directly communicates with the hardware of the computer
- **The Shell is the layer next to the Kernel and usually serves as the user interface**
- Common types of user interfaces in an operating system include Command-Line Interface (CLI) or Character User Interface (CUI) like in DOS, and Graphical User Interface (GUI) like in Windows
- **In a Command Line Interface (also called a Character User Interface) the user interacts with the operating system by typing commands on a command line**
- In a Graphical User Interface a window and menu based graphical interface is used which can be operated by using a pointing device like a mouse
- **The operating system maintains a directory or folder structure to store data in a computer storage media like a hard disk. The data is stored by the computer system in the form of files**
- A file is a collection of related data or information that is stored in a secondary storage media. A file can be either a data file or a program file
- **A data file basically stores data and can be numeric, alphanumeric, or binary in nature**
- A program file contains executable code and is also called an executable file or exe file
- **Each file is identified by a file name and an optional extension name which can be used to specify the type of the file**
- A file may also have certain file protection attributes like read-only, archive, system or hidden, which determine the type of file access given to the user
- **The various operations possible on a file include file creation, deletion, renaming, opening, closing, sorting, modification, copying etc.**
- **Transaction File is used to store input data temporarily until it is processed. For example a transaction file may contain the weekly data of employees in an organisation**
- Master File contains all the current data relevant to an application. Master files contain descriptive data which are permanent in nature, such as ID, name and address of an employee in an organisation, his gross salary etc.
- **Some applications use multiple programs for data processing. In such an application, the output produced by one program is used as an input for the next program. Here the temporary output of the first file is stored in a file called the output file**
- Data processing applications need to generate various types of reports based on user queries. A report file contains a soft copy of such a report
- **A backup file is used to keep a copy of a given file as a backup**
- In a serial file organisation records are stored in no particular logical sequence but stored serially one after the other
- **In a sequential file organisation, the records are stored in a particular order with respect to a given field of the record**
- In an indexed sequential file organisation the records are physically ordered with respect to a search key in the file. In addition to that, a primary index for the file is also maintained
- **In a direct file organisation the records in the file can be accessed directly using a special process on the search key of the file**
- Directories are used to group other directories or files and put them under a common heading.
- **Each directory can have a number of entries, one per file. When a file is created, a new entry for the file is created in the directory**
- Whenever a file is created, the operating system allocates the required number of clusters for storing the file depending upon the file size. Computers maintain on the disk, a table indicating the cluster location of a file. This table is known as the File Allocation Table or FAT and serves as a file index
- **The FAT is the most important area in a disk as it stores information regarding the location of a file and the availability of free space to store new files**

- The procedure of starting a computer by loading the basic components of the operating system into its ma memory is known as booting
- **The Power On Self Test (POST) is a diagnostic test that checks the state of the various components o the computer system**
- Depending upon the way a booting takes place, there are two different types of booting processes namely Wa Booting and Cold Booting
- **Warm booting skips the RAM test**
- Spooling or Simultaneous Peripheral Operations On-Line is a technique used to solve the problem of spe mismatch between the processor and peripheral devices like printers, keyboards etc.
- **A memory management scheme called Virtual Memory overcomes the problem of insufficient RAM by allowing the execution of processes using techniques like swapping and demand paging**
- Swapping is the process of transferring a block of data from the high speed on-line secondary storage to the ma memory (RAM) or vice versa
- **DOS is a single-user, single-tasking operating system that offers a command line character use interface through which the user has to type in the commands to execute them**
- The commands given to DOS are in general computer programs. Depending upon the location of the progra these are classified as Internal and External commands
- **The internal commands form a part of the COMMAND.COM program and include commands like CLS DATE, TIME, DIR, CD, MD, RD, COPY, DEL, RENAME**
- External commands on the other hand are command programs that are kept in the disk until they are needed an include commands like DISKCOPY, SYS, FORMAT, XCOPY, MOVE, BACKUP, RESTORE, DELTREE, ATTRIB
- **The scandisk utility checks the disk surface for any physical damage and tries to repair the disk surface first and in case it is unable to do so, it marks the damaged area as containing bad sectors in the FAT and prevents further writing of data on that portion**
- Before the disk can be used in a computer system it must first be prepared by a process called disk formatting
- **The pattern that is laid out during formatting, first divides the surface of a disk into a number of invisible concentric circles called tracks. Each track is again further subdivided into smaller sections called sectors during formatting**
- In Batch Processing, DOS allows to group several commands into a file with the extension 'bat' (for batch)
- **Microsoft Corporation, co-founded by Bill Gates introduced the first version of the Windows Operating System in 1985**
- Windows uses a GUI and allows users to run multiple applications at the same time using multitasking
- **The working space of the Windows system is known as the Desktop**
- Over the desktop are placed small pictures called icons of the different parts of the computer that one works wit like files, folders, recycle bin, network etc
- **At the bottom of the desktop is an area called the Taskbar. Whenever a program starts, an open window button for it appears on the Taskbar. The taskbar also displays the system time and program icons for quick launch of programs by a single click on them**
- When a program icon is clicked and the program is loaded into memory, the program appears in a rectangula frame called a Window on the screen
- **Each window contains a Title Bar across the top that indicates what the window contains. When a particular window is chosen, the title bar of that window gets highlighted**
- The region below the title bar represents the actual working area of the program. It contains a horizontal list o menus called the Menu bar. Below the Menu bar is a set of buttons that form the Toolbar
- **A window can also contain Scroll Bars to view the different parts of a program or file that are longer or wider than the size of the window**
- The registry is the place where Windows keeps most of its configuration information
- **To access any file or folder, one can click on the My Computer icon on the Desktop, when the My Computer window opens**
- Notepad is a simple text editor that can be used to type simple text in a given font style and size
- **Files or folders deleted by an user are not initially permanently deleted from the system. These are first stored in a special folder called the Recycle Bin**
- A program once installed cannot be removed simply by deleting the installed program file. It needs to be uninstalle both from the computer and from the Windows Registry
- **Disk Defragmenter utility can be used to analyse and defragment files in a secondary storage like a hard disk**
- The UNIX operating system was originally developed by a group of employees at the AT&T Bell Labs during the en of the 1960's
- **The UNIX OS was designed to be portable in nature which made it possible to be used in a wide variety of machine families than any other operating system**

- **The UNIX system uses a hierarchical file system, plain text for storing data, and treats devices as files**
- UNIX also uses the concept of piping whereby small programs can be lined together using a command line interpreter such that the output of a file acts as the input for the other to give the final output
- **Each account is identified by a login user name which has to be entered in response to the login prompt. In case the account is password protected, next UNIX will ask the user to type in the password**
- The superuser has access to all parts of the UNIX system. Accordingly, he can access all files and commands in the system. Superuser also looks after system configuration and setting up, maintaining, and deleting of user accounts
- **Once logged in, UNIX automatically places the user in his home directory as determined by the system administrator while creating the user account**
- Unlike DOS, the UNIX commands are case sensitive
- **Unlike DOS, filenames in UNIX do not have an extension. If any extension is used, it will be taken as part of the file name. Thus the user can use an extension for a file for his own convenience, but it is not a requirement for UNIX**
- Unlike DOS, UNIX has a standard directory structure
- **There are three types of file permissions, these are Read Permission, Write Permission, and Execute Permission**
- There are also three levels of permissions, these are Owner Level Permission, Group Level Permission, Others/World Level Permission
- **There are several shells available in UNIX. The user usually sticks to the shell that the system administrator has chosen for the user**
- The Bourne Shell was shipped with the original System-V release of UNIX in 1979 and has remained almost unchanged since then
- **The Korn Shell was developed by David Korn by adding several useful features over the functionalities of Bourne shell**
- The C Shell was developed by Bill Joy while working on the release of Berkeley UNIX (also called BSD). The C shell includes features like job history, aliases and some other features
- **Linux is an operating system that belongs to the UNIX family of operating systems. Before the advent of Linux, the powerful features of UNIX were not available for home computers**
- The name Linux comes from the Linux kernel that was originally written by Linus Torvalds
- **The first Linux kernel released to the public was Version 0.01 on May 14, 1991**
- The GNU Project was started by Richard Stallman in 1984
- **GNU programs are all released under a special licence called the GNU Public Licence which describes the principles of free, open-source software**
- By the term 'free', it means two things. Firstly, the user does not have to pay for the software, and secondly the user can have full access to the source code of Linux
- **Linux vendors and communities combine and distribute the kernel, GNU components, and non-GNU components, with additional package management software in the form of Linux distributions**
- For desktop systems usually a graphical user interface is used and KDE, GNOME, and Ubuntu are some of the most popular graphical user interfaces

Review Questions





Q1. **Multiple Choice Questions. Select from any one of the four options.** **1 each**

i) The hardware part of a computer can do any useful work only in the presence of:
a. software b. hard disk drive c. keyboard d. VDU monitor

ii) Software used to run the computer and manage the different resources of a computer:
a. Compiler software b. Translator software c. System software d. Interpreter software

iii) The most significant system software is the:
a. compiler b. loader c. linker d. operating system

iv) Operating System acts as a link between the:
a. hardware & user b. software & user c. hardware & software d. all of these

v) Programs used to do utility jobs like finding or compressing files, disk defragmenting etc. are:
a. library programs b. application programs c. general programs d. anti-virus programs

vi) Program used to translate an assembly language program to machine code is:
a. an interpreter b. an assembler c. a compiler d. an editor

vii) An interpreter is used to translate a program to machine code from:
a. C language b. high level language c. assembly language d. none of these

viii) A compiler is used to translate a program to machine code from:
a. BASIC language b. assembly language c. high level language d. none of these

ix) The output of a compiler is the machine language code also called the:
a. opcode b. object program c. machine program d. executable program

x) When a program is executed, it is broken down into a number of smaller units called:
a. segments b. units c. jobs d. objects

xi) Any new process entering a processing system must initially go to the:
a. running state b. process state c. blocked state d. ready state

xii) In case of an interrupt, the running state of a process is changed to:
a. occupy state b. stop state c. process state d. blocked state

xiii) The OS module that keeps track of the input/output requests from various processes:
a. file management b. device management c. data management d. link management

xiv) Operating system used for processing a set of jobs without any human intervention is called a:
a. Job Processing Operating System b. Batch Processing Operating Systems
c. Code Processing Operating System d. Data Processing Operating Systems

xv) To separate one job from another in a batch processing OS, special languages used are called:
a. job languages b. job set languages c. job control languages d. job data languages

xvi) An operating system that can handle multiple tasks together:
a. multi-variable operating system b. multi-processing operating system
c. multi-operation operating system d. multi-tasking operating system

xvii) An operating system that allow various users to share the CPU, memory and other resources of the computer system under its supervision is called a:
a. time sharing operating system b. time splitting operating system
c. time manoeuvring operating system d. time managing operating system

xviii) An operating system used in places where response time required for data processing is critical:
a. critical time OS b. response time OS c. real time OS d. special time OS

xix) The innermost layer which is the central controlling part of an operating system is the:
a. core b. shell c. kernel d. link

xx) The layer next to the kernel that usually serves as the user interface is called the:
a. shell b. core c. interface d. view layer

xxi) CLI stands for:
a. Command-Letter Interface b. Command-Line Interface
c. Code-Line Interface d. Computer-Line Interface

xxii) GUI stands for:
a. Graphical Utility Interface b. General User Interface
c. Graphical User Interface d. General Utility Interface

xxiii) A file is a collection of related:
a. operations b. pictures c. code d. data

xxiv) A program file that runs is also called:
a. a data file b. a code file c. an executable file d. a running file

xxv) File used to store input data temporarily until it is processed is called a:
a. Temporary File b. Transaction File c. Token File d. Test File

xxvi) A file that contains all the current data relevant to an application:

a. Data File b. Current File c. Transaction File d. Master File

xxvii) Records are stored in no particular logical sequence in a:
a. serial file b. sequential file c. transaction file d. direct file

xxviii) Records are stored in a particular order with respect to a given field of the record in a:
a. serial file b. transaction file c. direct file d. sequential file

xxix) In an indexed sequential file the records are physically ordered with respect to a:
a. data key b. super key c. sequential key d. search key

xxx) Files can be grouped together and put under a common heading called a:
a. class b. section c. directory d. group

xxxi) Area in a disk that stores information regarding the location of a file is called the:
a. MAT b. FAT c. SAT d. PAT

xxxii) The procedure of starting a computer by loading the basic components of the operating system into its main memory is known as:
a. listing b. starting c. loading d. booting

xxxiii) The diagnostic test that checks the state of various components of the computer system is:
a. COST b. HOST c. POST d. EOST

xxxiv) There are two different types of booting processes namely cold booting and:
a. warm booting b. hot booting c. general booting d. test booting

xxxv) The RAM test is skipped by:
a. hot booting b. general booting c. test booting d. warm booting

xxxvi) The technique that solves the problem of speed mismatch between processor and peripheral devices is called:
a. spooling b. scaling c. speeding d. matching

xxxvii) A memory management scheme that overcomes the problem of insufficient RAM is called:
a. virtual memory b. additional memory c. real memory d. ROM

xxxviii) The full form of DOS is:
a. disk opening system b. data operating system
c. daily operating system d. disk operating system

xxxix) DOS commands are classified as internal and:
a. data commands b. external commands c. virtual commands d. operating commands

xl) The internal DOS commands form a part of the program called:
a. utility.com b. system.com c. program.com d. command.com

xli) The utility that checks the disk surface for any physical damage and tries to repair it is called:
a. scanning disk b. scanner disk c. scan disk d. repair disk

xlii) Before a hard disk can be used in a computer system it must be prepared by a process called:
a. disk editing b. disk testing c. disk formatting d. disk checking

xliii) The pattern laid out during formatting of a hard disk first divides the surface of the disk into a number of invisible concentric circles called:
a. clusters b. sectors c. lines d. tracks

xliv) Each track is further subdivided into smaller sections called sectors during formatting:
a. clusters b. lines c. tracks d. sectors

xlv) The working space of the Windows system is known as:
a. Desktop b. Computer top c. Screen top d. Windows top

xlvi) Windows operating system keeps most of its configuration information in the:
a. system file b. registry file c. log file d. command file

xlvii) To access any file or folder, one can click on this icon on the Desktop:
a. Recycle Bin b. Networks c. Documents d. Computer

xlviii) Notepad is what type of software?
a. text editor b. photo editor c. audio editor d. video editor

xlix) A drawing and painting software supplied by Windows is called:
a. PhotoEdit b. Draw c. Paint d. Design

l) Files or folders deleted by an user are first stored in a special folder called:
a. Recycle Bin b. Recycle Area c. Recycle Bag d. Recycle Store

li) The utility used to analyse and rearrange files in a secondary storage like a hard disk is called:
a. disk rearranger b. disk editor c. disk monitor d. disk defragmenter

lii) The DOS command used to clear all the other commands already typed in the screen is called:
a. CLEAR b. CLS c. CLR d. CLEAN

liii) The DOS command used to list all the sub files or sub directories under a given directory is:
a. DIR b. LIST c. MENU d. LS

liv) The DOS command used to create a new directory or folder in DOS is called:
a. MD b. CD c. RD d. FD

lv) The DOS command used to delete a directory or folder in DOS is called:
a. MD b. RD c. CD d. DD

lvi) The DOS command used to move from one directory to another in DOS:
a. RD b. CD c. MV d. MD

lvii) CON is the device name used in DOS to indicate a console i.e. the:
a. mouse b. keyboard & monitor c. hard disk d. CD drive

lviii) The command in DOS that can be used to create a data file is:
a. control con b. copy con c. copy file d. file con

lix) The command *copy MyData.txt con* in DOS can be used to:
a. display a file b. print a file c. create a file d. rename a file

lx) The REN command in DOS is used to:
a. replace a file b. rename a file c. repair a file d. read a file

lxi) The DEL command in DOS is used to:
a. delete a file b. delete a directory c. delete a file or folder d. all of these

lxii) The DOS command used to view a listing of files & folders in an easy to read graphical format:
a. view b. graph c. tree d. display

lxiii) The DOS command used to move a file from one location to another:
a. ren b. move c. copy d. cd

lxiv) The DOS command *MOVE MyDocs Document* is used to:
a. delete a file b. rename a file c. move a file d. copy a file

lxv) The DOS command that helps to set the attribute for a file is:
a. attb b. attr c. attribute d. attrib

lxvi) The DOS command used to display the disk or drive space specified is:
a. vlm b. volume c. vol d. vm

lxvii) The DOS command used to check a disk for errors and display a status report, is:
a. checkdisk b. chkdsk c. checkdsk d. chkdisk

lxviii) A data file can be created or edited in DOS using the command:
a. attrib b. rename c. create d. edit

lxix) The wildcard character symbol used to specify a group of characters in a file name is:
a. # b. ? c. $ d. *

lxx) The wildcard character symbol used to specify a single character in a file name is:
a. ? b. * c. # d. $

lxxi) Which one of the following is not a type of file permission in UNIX?

a. write b. copy c. read d. execute

lxxii) Which one of the following is not a level of file permission in UNIX?
a. world b. owner c. user d. group

lxxiii) Which one of the following commands is used to view the directory listing in UNIX?
a. list b. dir c. show d. ls

lxxiv) Which one of the following commands is used to remove only directories in UNIX?
a. rm b. del c. rd d. rmdir

lxxv) Which one of the following commands can be used to remove both directories and files in UNIX?
a. remove b. rm c. rmdir d. del

lxxvi) Which one of the following commands can be used to rename files in UNIX?
a. ren b. mv c. rename d. cp

lxxvii) Which one of the following is not a wild card character used in UNIX?
a. [ ] b. * c. ? d. _

lxxviii) Which one of the following is not a type of shell in UNIX?
a. Sea b. Job c. Korn d. Bourne

lxxix) The process where the output of a command serves as the input of another command in UNIX:
a. flow b. channel c. i/o d. piping

lxxx) The text editor name used in UNIX:
a. vi b. edit c. edlin d. notepad

Q2. **Short Answer type questions:** **1 each**



i) What do you mean by system software?
ii) What is the use of a text editor?
iii) What is an assembler?
iv) What is an interpreter?
v) What is a compiler?
vi) State one difference between a compiler and an interpreter.
vii) What do you mean by application software?
viii) What category of software are a compiler and an interpreter?
ix) State the name of any two important functions of an operating system.
x) State the use of the device management module of an operating system.
xi) What is a multitasking operating system?
xii) What is a kernel with respect to an operating system?
xiii) What is a shell with respect to an operating system?
xiv) Write the full form of POST in connection with booting.
xv) Write the full form of GUI.
xvi) Write the full form of CLI.
xvii) State one difference between a command line and a graphical user interface.
xviii) Write the name of any two properties of a file.
xix) What is the use of a file extension?
xx) Name any two different types of files.
xxi) What types of file is indicated by the file extension .exe?
xxii) What is a transaction file?
xxiii) What is a master file?
xxiv) Name any one type of file organisation.
xxv) State one difference between serial and sequential file organisation.

xxvi) State one difference between a file and a folder.
xxvii) State one use of a file allocation table.
xxviii) What do you mean by warm booting?
xxix) State one reason for using virtual memory.
xxx) State one difference between internal and external commands in DOS.
xxxi) Name any two internal commands in DOS.
xxxii) Name any two external commands in DOS.
xxxiii) Name any two types of switches that can be used with the DIR command in DOS.
xxxiv) Name any two switches used with the ATTRIB command in DOS.
xxxv) State one difference between the COPY and MOVE operations in DOS.
xxxvi) State one difference between the RMDIR and DEL operations in DOS.
xxxvii) State one use of the COPY CON command in DOS.
xxxviii) What do you mean by a track with respect to a hard disk?
xxxix) What do you mean by a sector with respect to a hard disk?
xl) What do you mean by a wildcard character in DOS?
xli) Write the symbols used for wildcard characters in DOS.
xlii) Write the DOS command statement used to display the directory and file listing of the directory *WORKS* under the *D:* drive.
xliii) Write the DOS command statement used to make a directory called *PERSONAL* under the *HOME* directory located under the *C:* drive.
xliv) Write the DOS command statement used to remove a directory called *LETTERS* from under the *PERSONAL* directory located under the *E:* drive.
xlv) Write the DOS command statement used to move the file *poetry.txt* from the *TEMP* directory under the *C:* drive to the *D:* drive.
xlvi) Write the DOS command statement used to copy the file *myCV.docx* located under the *TEMP* directory in *D:* drive to the *C:* drive.
xlvii) Write the DOS command statement used to create a text file called *poetry.txt* under the *D:* drive using the *COPY* command.
xlviii) Write the DOS command statement used to display the contents of the text file *biodata.txt* stored in the *D:* drive.
xlix) Write the DOS command statement used to print the contents of the file *poetry.txt* in the *C:* drive using the printer connected to the parallel port with the device name *LPT1*
l) Write the DOS command statement used to rename the directory *MyWorks* under the *D:* drive to *Personal*.
li) Write the DOS command statement used to delete the file with the file name *extra.txt* located under the useless folder under the *E:* drive.
lii) Write the DOS command statement used to change the attribute of the file *MyCV.docx* to a read-only, archive file.
liii) Write the DOS command statement used to copy all files with the file extension *.tmp*, irrespective of their name, from the *D:* drive to the *Temp* directory located under the *C:* drive.
liv) Write the DOS command statement used to display all files whose name starts with the letters *work* and which have a three letter extension name (like .txt etc.) from the *D:* drive.
lv) What do you mean by the Desktop in windows?
lvi) What is the taskbar?
lvii) What is the Recycle Bin?
lviii) Which key combination can delete a file in Windows without sending it to the Recycle Bin?
lix) State one utility of the Disk Defragment operation in Windows.
lx) Write the UNIX command statement used to display the directory and file listing of the directory *progs* under the *usr* directory.
lxi) Write the UNIX command statement used to make a directory called *works* under the *usr* directory.

lxii) Write the UNIX command statement used to remove a directory called *tempo* from under the *mywork* directory under the *usr* directory.

lxiii) Write the UNIX command statement used to move the file *eassy.final* from the *temp* directory under the *usr* directory to the *final* directory under *usr* directory.

lxiv) Write the UNIX command statement used to rename a file *prog01.c* stored in the *usr* directory to *factorial.c*.

lxv) Write the UNIX command statement used to join the contents of two files *file1.poetry* and *file2.poetry* to form a new file called *submit.poetry*. All the files are stored under the *works* directory under *usr* directory.

Q3. **Long Answer type questions:** **7 each**

i) Explain any three functions of an operating system. State one difference between a command line and a graphical user interface. 6+1

ii) State any four important differences between an interpreter and a compiler. Write a short note on the kernel of an operating system. 4+3

iii) What do you mean by system software and application software? Names any three system software and briefly state their use. 4+3

iv) Write briefly on any three different types of file organisation. Write the name of any one type of data file. 6+1

v) Briefly describe the utility of the device management module of an operating system. What do you mean by a transaction file and a master file? 3+2+2

vi) Briefly explain the terms kernel and shell with respect to an operating system. Write the names of any two external commands in DOS. 3+3+1

vii) What is a batch processing operating system? State any four properties of a file. What is the use of a file extension? 3+2+2

viii) What do you mean by booting? State the difference between cold booting and warm booting. What is virtual memory? Name any one internal DOS command. 2+2+2+1

ix) What is a file allocation table? What is the use of virtual memory? State the difference between a file and a folder. What type of file is indicated by the file extension .exe? 2+2+2+1

x) What do you mean by a real time OS and a time sharing OS? Explain the meaning of the term spooling with respect to an operating system. 2+2+3

xi) Explain the meaning of the terms 'multiprocessing' operating system and 'multitasking' operating system. Write a short note on the concept of virtual memory. Name the UNIX command used to rename a file. 4+2+1

xii) Explain the meaning of the term spooling with respect to an operating system. What is virtual memory? What type of file is indicated by the file extension .exe? 3+3+1

xiii) What do you mean by a real time operating system? State the difference between Internal and External commands in DOS with proper examples. 3+4

xiv) Name two internal commands and two external commands in DOS. Name and explain any two types of switches that can be used with the DIR command in DOS. What is the use of the Recycle Bin? State one difference between the copy and cut operations on a file. 2+2+2+1

xv) What is the use of the ATTRIB command in DOS? What is the difference between the COPY and MOVE operations in DOS? What is the utility of the COPY CON statement in DOS? Name the file extension associated with a file that can be run. 2+2+2+1

xvi) What do you mean by formatting a magnetic storage disk? What are tracks and sectors with respect to a magnetic storage disk? What do you mean by wild card characters in an operating system? How can you recover a deleted file in the Windows operating system? 2+2+2+1

xvii) Write the DOS command statement used to move the file cost.txt from the TEMP directory under the C: drive to the WORK directory under the directory EXPENSE in the D: drive. Discuss any two methods of copying a file from one folder to another in the Windows operating system. What is the use of the Internet Explorer? 2+2+2+1

xviii) Write the DOS command statement used to copy the file movie.dat located under the MyMovies directory under the D: drive in the same place, but with a different file name. Create a text file using the COPY command to write your name and address in the file called biodata.txt. State any three utilities of the Windows Explorer. 2+2+3

xix) Write the DOS command statement used to move the file MyCV.doc from the sub-directory Word under MyDocs in the D: drive to the directory Backup in the E: drive, but with the changed file name Biodata1.doc. State any two methods of creating a folder in the Windows operating system. Write any three utilities of using the Computer (or My Computer) option in the Windows operating system. 2+2+3

xx) Write the DOS command to copy all files with the file extension .docx, irrespective of their name, from the Question directory under the D: drive to the Temp directory located in the Personal directory under the E: drive. State any two differences between a CUI and a GUI. Name any three methods of copying a file from one folder to another in the Windows operating system. 2+2+3

xxi) State and explain any two methods of creating a folder in Windows OS. How can you rename a folder in Windows? What is the Recycle Bin in Windows? 2+2+2+1

xxii) How can you recover a deleted file in Windows? How can you view the memory size of the hard disk of your computer? What is a Jump List? Write the name of the DOS command used to view the file listing in a graphical manner? 2+2+2+1

xxiii) How can you change the date and time in Windows OS? How can you search for a given file in the Windows operating system? What is the full form of the file extension .pdf? 4+2+1

xxiv) Explain the Aero Shake and Aero Snap options in Windows OS. Write a short note on the vi editor in UNIX. 4+3

xxv) What do you mean by a 'piping' operation in UNIX? State any two important characteristics of the UNIX OS. State the difference between the rm and rmdir commands in UNIX. What is the use of the cal command in UNIX? 2+2+2+1

xxvi) Explain the chmod command in UNIX. What is the utility of the piping operation in UNIX? State one use of the cat command in UNIX. 4+2+1

xxvii) Discuss the different types and levels of file permission in UNIX. Which command can be used to rename a file in UNIX? 6+1

xxviii) Write a short note on any two standard shells used in UNIX. What do you mean by the term 'superuser' in UNIX? Name any two standard directories used in UNIX. 4+2+1

xxix) State the use of the cat command in UNIX with the help of examples. Explain the three levels of file permission in UNIX. Explain the different types of wild card characters used in UNIX. 2+3+2

xxx) How can you change the date and time in Windows OS? How can you search for a given file in the Windows operating system? What is the full form of the file extension .pdf? 4+2+1

CHAPTER 7
Algorithm and Flowchart

- General Concepts 7-1
- Different Phases of Programming 7-1
- Algorithm 7-3
- Flowchart 7-5
- Pseudo Code 7-7

7.1 General Concepts

Programs are a set of codes that take some input and produce an output. Though the code generated depends upon the type of (computer) language used, however the **procedure should be a set of general rules** to solve the given problem and should be **independent of the language used**. This general rule is meant to solve a given problem stepwise, based on some procedure or logic. The language is a tool to implement the rules. For example to prepare a cup of tea, the general procedure is:

1. Put some water in a container, light a stove and put the container on it
2. Check if the water is boiling
3. If not, wait for the water to boil and again go to step 2, else go to step 4
4. Put off the stove and add some tea leaves and wait for 2 minutes
5. Pour the liquor into a cup, add sugar and milk and serve

The above result can also be achieved using a pan or a kettle, a gas burner or an oil stove, or powdered milk instead of dairy milk etc. In each case though the general procedure remains the same but the actual procedure and details of getting the result may vary.

As a second example take the problem of calculating the result of **y = 3 x 4 / 2 + 4^(3 – 1)**

To get the result we have to start the calculation from the left and follow the BEDMAS rule as shown below:

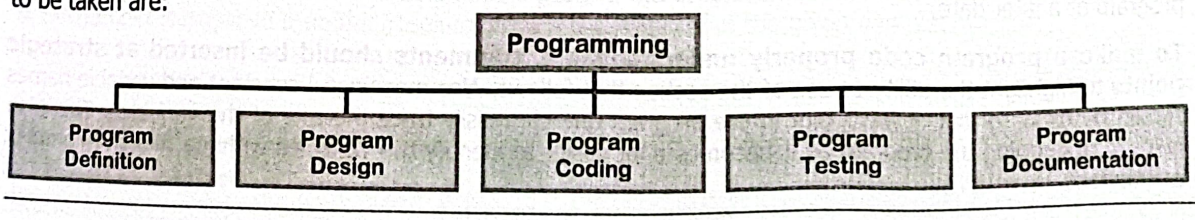
1. Look for the presence of any **Brackets**
2. If brackets are present, calculate the part inside the bracket, go to step 1
3. Look for any **Exponentiation** (power) operation
4. If an exponentiation operation is present, perform the exponentiation, go to step 3
5. Look for any **Division** operation
6. If a division operation is present, perform the division, go to step 5
7. Look for any **Multiplication** operation
8. If a multiplication operation is present, perform the multiplication, go to step 7
9. Look for any **Addition** operation
10. If an addition operation is present, perform the addition, go to step 9
11. Look for any **Subtraction** operation
12. If a subtraction operation is present, perform the subtraction, go to step 11

The value of the expression can be calculated correctly by **following the above set of rules in the order written**. If the order is changed, then the output will be different. Thus the order of the steps is important in such a procedure. Moreover we have completed the calculation in a **finite number of steps**. Once this rule has been fixed, we can perform the calculation either manually, or using a calculator or using a computer program, i.e. by using different means. However, whatever be the means, the procedure remains the same and independent of the means.

7.2 Different Phases of Programming

To analyse a problem and then to convert it to a computer program to solve the problem, the steps that need to be taken are:

Different phases of programming





Program Definition Phase

- **The Program Definition Phase:**

This is the first step in the programming process. In this step we work out **what must be done** to solve the problem and try to extract from the problem statement the different inputs in the form of **variables and constants**, along with any **special formula** or **relationship** that may be required in solving the problem to get the output. The form of the **desired output** should also be worked out.

As an example, consider the familiar problem of **finding the roots of a quadratic equation**. Here the **'what'** involves first identifying the input variables, namely the **coefficients** of the variable term and the constant term. The next requirement is to know whether any special formula or relationship is required. In this case the Sridharacharya relationship should be used. Finally we have to know the types of outputs that may arise i.e. real-equal, real-unequal or imaginary roots.



Program Design Phase

- **The Program Design Phase**

Next comes the **'how'** part. Since there can be more than one way to solve a problem it is not always easy to find the best solution to a particular problem. As for the **'how'** part of the quadratic equation problem first we have to find the **discriminant** and then check whether its value is zero, greater than zero or less than zero. Then depending upon the value of the discriminant we have to take the square root of the discriminant and find the roots using the Sridharacharya relationship.

Sometimes it may be helpful to assume the solution and start working backward to the starting condition. In case the final solution is not known even a guess at the solution can be helpful. For example to find the square root of 5 we assume the square root to be a number greater than 2 (as $2^2=4$) and less than 3 (as $3^2=9$) and square that number to find how close it is to the exact root.

Some important concepts to implement the logic in the problem design phase include the development of an **algorithm, flowchart, or a pseudo-code** from the problem definition. In the following sections we will have a detailed discussion on the above methodologies.



Program Coding Phase

- **The Program Coding Phase:**

Once the algorithm, flowchart or pseudo-code for a problem has been developed, then the logic to solve the problem has been established. The next phase is the actual coding of the program i.e. to translate the logic into program code in a step by step manner using a particular programming language. Depending on the choice of the programming language the **program syntax will be different, though the main logic and structure remains the same.**

Care should be taken so as to **minimise syntactical and logical errors**. The compiler or the interpreter can help you identify syntactical errors during the time of compilation or interpretation. But it is sometimes very difficult and time consuming to detect and debug a logical error.



Program Testing Phase

- **The Program Testing & Debugging Phase:**

After the program has been written we have to **check if the program runs correctly** i.e. whether we get the desired outputs. For a good program, even if invalid inputs are supplied the program should be able to detect them. **All sets of inputs or input conditions are to be given** to check if they produce proper outputs. If the program does not run exactly as expected then try to find out if there are any syntactical errors, whether you have missed out on any punctuation, used any reserved word as a variable name etc. Next try to find out if there are any **logical errors**. To tackle such problems try to follow the execution of the program step by step. Working out the program by hand (dry run) before attempting to execute it with a valid input data, can also be helpful.



Program Documentation Phase

- **The Program Documentation Phase:**

Program documentation should also be considered as an integral part of programming. Proper documentation can help a person to understand the logic of a program by going through the listing. This can help both the person who has designed the program and any other person who wants to inspect the source code of the program at a later date.

To make a program code properly understandable, comments should be inserted at strategic points to highlight the utility or use of the section that follows. Also meaningful constant and variable names should be used so that at any point in the program one can easily trace the role of the variables. This also helps in debugging the program as it becomes a lot easier to identify the variables within a maze of complex calculations.

7.3 Algorithms

Similarly, to solve a particular problem using a computer program we have to follow a specific set of rules and write the code in a particular computer language to implement the steps. To design the logic of a program several tools may be used, like the development of an **algorithm** from the problem definition.

An **Algorithm** is a **sequence of precise and unambiguous instructions** designed in such a way that if they are executed in the specified sequence the desired result is obtained within a **finite number of steps**.

Definition of Algorithm

In short the **steps** that need to be followed **to achieve the desired result** form the algorithm. It should be:

- **Effective**, which means that an answer should be found when the algorithm is applied, *and it*
- **Finishes,** that is, it **has a finite number of steps** and should **not go on infinitely**

An Algorithm is a sequence of precise and unambiguous instructions to obtain the solution to a problem in a finite number of steps.

As an example take the problem of **finding the average of 'n' numbers**. The program can be written for a **set of known & fixed numbers** or the program can be designed for a **variable number of inputs**. We discuss below the algorithm for both the cases.

The steps for finding the average of a **fixed number of inputs** are:

Step1: Add the numbers ($n_1 + n_2 + n_3 +, \ldots, + n_n$) to get the sum **S**
Step2: Divide the sum by the number count i.e. n to get the average **Avg := S/n**
Step3: Print the average **Avg**

In an algorithm, the symbol **:=** indicates an **assignment or store operation** i.e. the value **S/n** is stored in the variable **Avg**. In contrast, the **=** symbol is used to **compare two values**, as shown later.

The steps for finding the average for **variable number of inputs** are:

Step1: Initialise a variable called **Sum** to zero i.e. **Sum := 0**
Step2: Initialise a counter **C** to 0 to count the number of inputs, i.e. **C := 0**
Step3: Ask for the input **Inp**
Step4: Add the input value to the existing value of the variable **Sum** i.e. **Sum := Sum + Inp**
Step5: Increase the count value by 1 i.e. **C := C + 1**
Step6: If more numbers are to be input, then go to **Step3**, else go to **Step7**
Step7: Calculate the average **Avg** as the **Sum** divided by the count **C** i.e. **Avg := Sum/C**
Step8: Print the average **Avg** and terminate program

In the first case the program design is specific and each time the numbers are changed, the program also needs to be changed to accommodate the new numbers. In the second case the program is a general one and the user is able to input any number of inputs to find the average.

It can be seen that the second algorithm is a little complicated than the first but it clearly states the steps that need to be taken in case we want to prepare a general program to carry out the average and hence is a better choice. Thus the same problem can have different algorithms and different procedures.

Examples of Algorithms

Example-1: Algorithm to add two numbers:

Step1. Input **a**
Step2. Input **b**
Step3. **sum := a + b**
Step4. Print **sum**
Step5. Stop

Example-2: Algorithm to derive the absolute value of any number.

To derive the absolute value of a number (i.e. the positive magnitude of a value) we have to first check if the number is positive or negative. If the number is positive the absolute value is the same as the number itself. If the number is negative then the absolute value is the negative of the given number.

Step1. Input **num**
Step2. If **num < 0**, then
  a. **num := (–1) * num**
Step3. Print **num**
Step4. Stop

Example-3: Algorithm to check if a number is even or odd:

A number is even if it is evenly divisible by 2 and hence produces a 0 remainder, else it is odd. In that case a remainder of 1 is obtained. **Modulo** operation is used to get the **remainder of dividing a number X by another number Y**. Thus 14 (modulo) 3 is 2, as the remainder of the division is 2.

Step1: Input **num**
Step2: If **num (modulo) 2 <> 0**, then [the symbol **<>** is used to denote 'not equal to']
  a. **Print** "Odd Number"
Step3: Else
  a. **Print** "Even Number"
Step4: Stop

Example-4: Algorithm to find the percentage of marks obtained in Physics, Maths, and Computer Science by a student, out of a total of 300 marks.

Step1. **sum := 0** (*variable sum assigned value 0*)
Step2. Input **Phy** (*input marks of physics*)
Step3. **sum := sum + Phy** (*Add marks of physics to sum*)
Step4. Input **Math** (*input marks of math*)
Step5. **sum := sum + Math** (*Add marks of math to sum*)
Step6. Input **Coms** (*input marks of computer science*)
Step7. **sum := sum + Coms** (*Add marks of computer science to sum*)
Step8. **Percentage := sum*100/300**
Step9. Print **Percentage**
Step10. Stop

Example-5: The same problem can also be done for a variable number of subjects, using a counter:

Step1. **sum := 0** (*Declare variable for sum calculation and initialise it to 0*)
Step2. **count := 0** (*Declare variable for counting number of subjects and initialise it to 0*)
Step3. Input number of subjects: **num** (*Input the total number of subjects to use*)
Step4. Input marks of a subject: **marks** (*Input marks for a given subject*)
Step5. **sum := sum + marks** (*Add marks to sum total*)
Step6. **count := count + 1** (*Increase count value by 1*)
Step7. If **count <> num** then (*If count value is not equal to num then loop*)
  a. Go to **Step4**
Step8. **percentage := [sum/(num*100)] *100**
Step9. Print **percentage**
Step10. Stop

The above algorithm shows the **use of a 'loop'**, i.e. a section of the code that gets repeated (**steps 4 to 7**).

Example-6: A shopping mall is giving discount on items purchased. If the purchased item is a shirt, then 10% discount is given. If the item is a trouser, then discount given is 20%. If item is a cap then 5% discount is given. Write down the algorithm for the net purchase amount.

Step1: Input item type: **Item**
Step2: Input item price: **Price**
Step3: If **Item = "shirt"** then
  a. **Amount := Price – Price * 10 / 100**
Step4: If **Item = "trouser"** then
  a. **Amount := Price – Price * 20 / 100**
Step5: If **Item = "cap"** then
  a. **Amount := Price – Price * 5 / 100**
Step6: Print **Amount**
Step7: Stop

Note the use of the '=' operator to **compare the value stored in item** with other values.

Example-7: Let us now write down the algorithm to generate the series 1, 2, 3, 4, 5, 6 ...

The series is generated by adding 1 to the previous term in the series, starting from 1.

Step1. **count := 1** (*Declare counter and initialise it to 1*)
Step2. Input the total number of terms to print: **N**
Step3. Loop while **count <= N** (*Continue loop as long as count<=N*)
  a. Print **count**
  b. **count := count + 1** (*Increment count value by 1*)
Step4. Stop

To generate the series we have **used a different form of the loop statement**. Note that the steps **3a** and **3b** get repeated as long as **count** value is less than or equal to **N**. This is expressed by using the '**Loop while**' statement. The steps 3a and 3b loop as long as the condition **count<=N** is true. The moment the condition becomes false, the loop ends and stops repeating the statements 3a and 3b.

Example-8: Algorithm to generate the series 1^2, 2^2, 3^2, 4^2, 5^2, 6^2 ... and get its sum up to N terms.

The problem is similar to the previous one, however instead of the count value, the **square of the count value** is getting added here.

Step1: **count := 1** (*Declare counter and initialise it to 1*)
Step2: **sum := 0**
Step3: Input the total number of terms to add: **N**
Step4: Loop while **count <= N** (*Continue loop as long as count<=N*)
  a. **sum := sum + count*count**
  b. **count := count + 1** (*Increment count value by 1*)
Step5: Print **sum**
Step6: Stop

Example-9: Let us now write down the algorithm to print the first N odd numbers i.e. 1, 3, 5, 7, 9 ...

Step1: **count := 1** (*Declare variable for counting number of terms and initialise it to 1*)
Step2: **Term := 1** (*Declare variable for series term and initialise it to first term value i.e. 1*)
Step3: Input the total number of terms to print: **N**
Step4: Loop while **count <= N** (*Continue loop as long as count<=N*)
  a. Print **Term**
  b. **Term := Term + 2** (*Add 2 to current value of term to get the next term value*)
  c. **count := count + 1**
Step5: Stop

When an algorithm is expressed in a pictorial manner with special symbols to indicate the different types of instructions, then it is called a Flowchart.

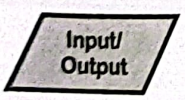
7.4 Flowcharts

When an **algorithm is expressed in a pictorial manner** with special symbols to indicate the different types of instructions, then it is called a **flowchart**. The actual instructions are written inside boxes of different shapes indicating different functions. Solid lines with arrows indicating the flow of programs are then used to connect the boxes. The different symbols which are used for a flowchart include:

Definition of Flowchart

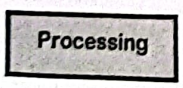


The Terminal denotes the **START**, **STOP** or **HALT** in the program logic flow. It is the first and the last symbol in a program. Halt can be used when there is an error condition in the program.



Any **input or output operation** from an input/output device is shown by this symbol. The input can be from the keyboard, disk, tape, mouse etc. while output can be to the monitor, printer etc.

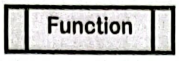
Shapes used with a Flowchart



This symbol is used to represent **arithmetic and data movement operations**. All arithmetic processes (add, subtract, multiply, divide etc.) and logical process of moving the data from one location to another are denoted by this symbol.



This symbol is used at a point where **a decision is to be made** and a **branching is necessary** depending upon the decision. The criteria for the decision should be indicated inside the box and the number of paths to follow should also be mentioned.



This symbol is used **to indicate any predefined function in a flowchart**. The function name may be defined by a separate flowchart or an algorithm or can be any standard function.



This is the symbol of a **Connector**. Whenever a flowchart becomes too long and is difficult to fit within a given page, it can be broken down into parts and the parts are joined by Connectors **to maintain continuity**. This symbol thus signifies an entry to or an exit from one part of the flowchart to another.



Flow-lines with arrowheads are used to **indicate the flow of data** or an operation. It determines the exact sequence in which the program flows.

Advantages of using Flowcharts

The **advantages** in using a flowchart for planning a program are stated below:

a. Since flowcharts are not based on a particular programming language, once a logical solution of a problem has been charted it can be **implemented in any programming language**.

b. Being pictorial in nature, flowcharts are **easy to understand**.

c. Large programs can be broken down into smaller modules and different individuals entrusted with working out a particular module. A main flowchart showing the **interconnectivity of the different modules** will help to put together the final program.

d. **Error** or bug in a program can be **easily detected** from a graphical flowchart.

In spite of these apparent advantages in using flowcharts, there are some obvious **disadvantages** in using flowcharts. Some of them are stated below:

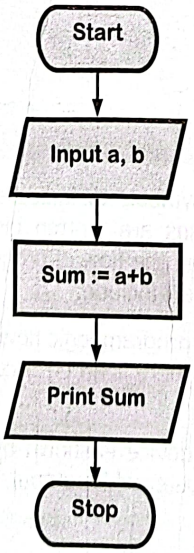
Disadvantages

a. Being pictorial in nature, flowcharts are **time consuming to draw**. When writing small programs it may be easier to draw flowcharts but for large programs it may becomes very inconvenient.

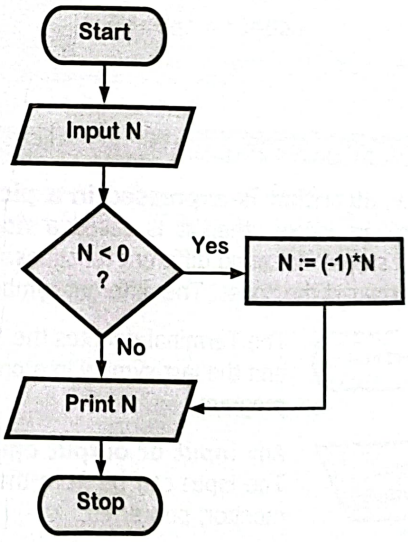
b. In case any **change** needs to be incorporated, it is not easy to do the same in an existing flowchart but the **flowchart needs to be redrawn** and this is a time consuming process.

Examples of Flowcharts

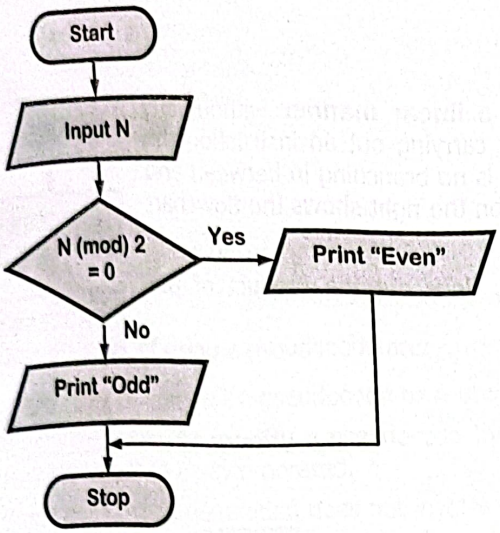
Example-10: Add two numbers.



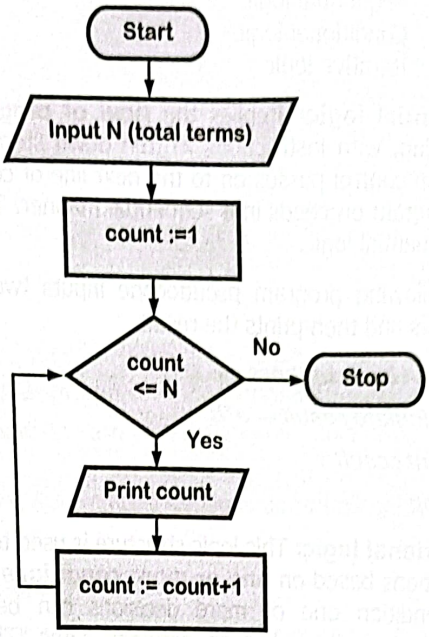
Example-11: Get the absolute value of any number. (Use of branching operation with a decision box)



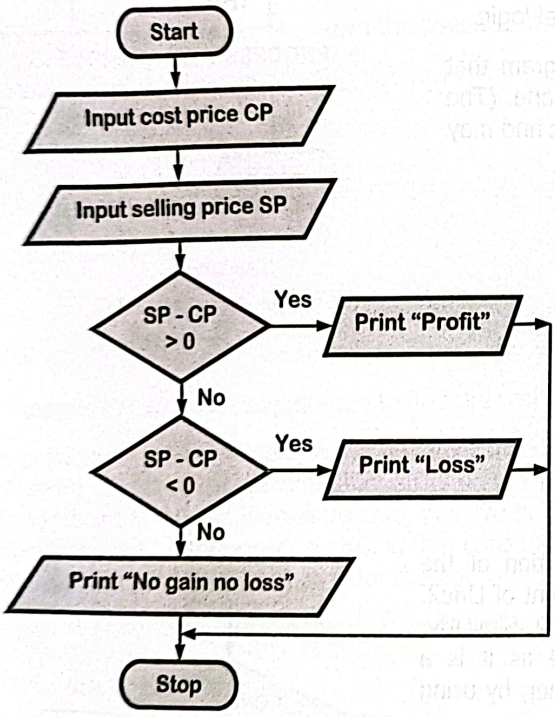
Example-12: Check if number is even or odd.



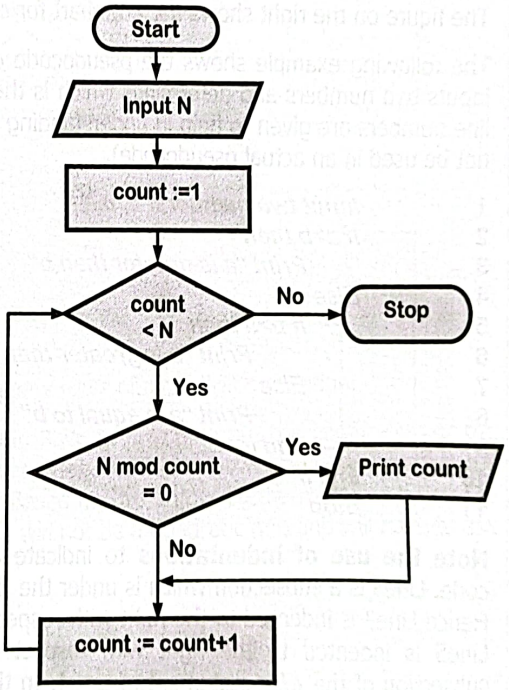
Example-13: Print the series 1, 2, 3, 4, 5, 6 ... (Use of loop operation, using a decision box)



Example-14: Check if profit or loss is made. during buying/selling items.



Example-15: Print the factors of a given number.



When an algorithm is written in a structural form that resembles computer instructions it is called a Pseudocode.

7.5 Pseudocode

To overcome the disadvantages of a flowchart the technique of pseudo code can be used. Pseudo means **imitation** and code means the **instructions** for the program. The pseudo or false instructions are written in a common language in a **structural form** that **resembles computer instructions**.

Since a pseudo code is similar to a computer program in structure but written in a common ordinary language it is sometimes called **Program Design Language (PDL)**. The pseudo code is thus a **step in between** the algorithm and the actual program code.



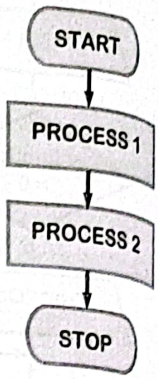
Pseudocode

The basic logic structures that are used to write a pseudo code and that have been found to be sufficient in writing almost all computer programs are:

1. Sequential logic
2. Conditional logic
3. Iterative logic

Sequential logic

Sequential logic: Implies the **flow of program** is in a **linear manner** without any branching, with instructions written down stepwise. After carrying out an instruction the program control passes on to the next line of code. There is no branching in between and the program proceeds in a sequential manner. The figure on the right shows the flowchart for sequential logic.

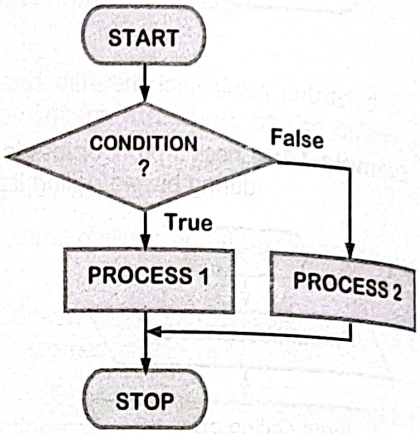


The following program pseudocode inputs two numbers, finds out the product of the numbers and then prints the result.

```
Input two numbers a & b
Calculate result ← a*b
Print result
Stop
```

Conditional logic

Conditional logic: This logic structure is used to perform a set of instructions based on **one or more conditions**. Depending upon the condition one or more decisions can be taken and the program logic branches to different paths. The **If-Then-Else** structure or **If-Else** structure is used to depict such a situation. **If** a certain condition is satisfied **then** the program carries out a set of instructions **else** another set of instructions are carried out. The figure on the right shows the flowchart for conditional logic.



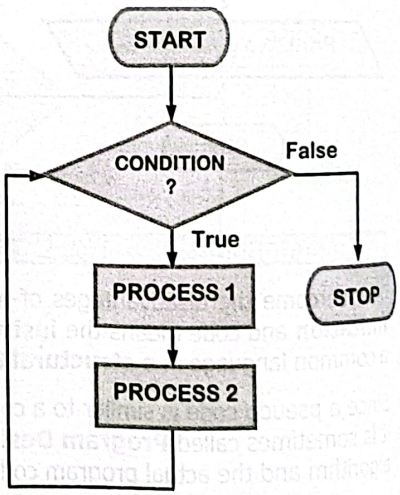
The following example shows the pseudocode of a program that inputs two numbers and determines which is the larger one. (The line numbers are given to help in understanding the logic and may not be used in an actual pseudocode).

```
1   Input two numbers: a, b
2   If a>b then
3       Print "a is greater than b"
4   Else
5       If b>a then
6           Print "b is greater than a"
7       Else
8           Print "a is equal to b"
9       End if
10  End if
11  Stop
```

Note the use of indentations to indicate a subsection of the code. Line3 is a subsection which is under the ***if*** statement of Line2. Hence Line3 is indented to the right with respect to Line2. Similarly, Line5 is indented to the right with respect to Line4 as it is a subsection of the ***else*** statement of Line4. In this manner, by using indentations, each **logical block is identified**.

Iterative logic

Iterative logic: When **one or more instructions** need to be **executed several times** based on certain conditions then we get iterative logic. The instruction block that needs to be repeated is put inside the body of the loop and the looping continues as long as a certain condition is satisfied. The **While**, **Repeat-Until**, and **For-Next** structures are usually used to indicate a loop in a pseudocode. The general structure of an iterative loop is shown by the flowchart on the right.



In the following example the pseudocode is used to calculate the **factorial of an input number.**

```
Input the number: num
Define: factorial ← 1
Initialise loop counter: I ← 1
While I<= num, loop
    factorial ← factorial * I
    I ← I+1
End loop
Print the value of factorial
Stop
```

The **advantages** of using a pseudocode are:

Advantages of using pseudocode

1. It is **easier to convert** a pseudocode **to a programming language** code than from a flowchart.
2. It is a lot **easier to modify** a pseudocode than a flowchart, in case some change in logic or additional features needs to be incorporated.
3. Since pseudocode generation does not involve any graphics, it is **much less time consuming** to write a pseudocode than to draw a flowchart.

In spite of the above advantages there are certain **disadvantages** which include:

Disadvantages

1. **Graphic representation is much easier to interpret** but pseudocode does not have any such scope.
2. Pseudocode generation **does not involve any standard rules** and hence **different programmers have their own style of writing** a pseudocode resulting in a lack of uniformity.

Examples of pseudocode

Example16: Let us now write down the pseudocode to reverse the digits of a number.

```
Define variables: num, rev_num ← 0
Input: num
        While num ≠ 0
            rev_num ← rev_num*10 + num(MOD)10
            num ← INT(num/10)
        End loop
Print rev_num
Stop
```

Example17: Write the pseudocode to find the real and imaginary roots of a quadratic equation.

In dealing with this problem as we have discussed earlier, we have to take as input the coefficients of the different terms of the quadratic equation of the form $ax^2+bx+c=0$. Then we have to check whether the discriminant $D=b^2-4ac$ is equal to zero, positive or negative. Based on the sign of D, the roots will be real or imaginary. We have to check if a=0, in that case the equation will not be a quadratic one and will have to ask for a fresh input. The pseudocode for the above problem is:

```
Input the coefficients: a, b, c while a≠0
d ← b² – 4*a*c
If d = 0
        root1 ← –b/(2*a) ,  root2 ← –b/(2*a)
        Print root1, root2
If d > 0
        root1 ← –b/(2*a)–(√d)/(2*a), root2 ← –b/(2*a)+(√d)/(2*a)
        Print root1, root2
If d < 0
        d ← –d
        root1 ← –b/(2*a)+ I (√d)/(2*a), root2 ← –b/(2*a) – i (√d)/(2*a)
        Print imaginary roots: root1, root2
Stop
```

Example18: Write down the pseudocode to read 10 real numbers and find their average.

```
Define an array to hold 10 real numbers: Arr(10)
Define variables: i ← 1, sum ← 0, average
While i <= 10
    Input Arr(i)
    i ← i+1
End loop
i ← 1
While i <= 10
    sum ← sum + Arr(i)
    i ← i+1
End loop
average ← sum/10
Print average
Stop
```

Example19: Write the pseudocode to reverse the order of the numbers in an array of numbers.

```
Define variables: i ← 1, temp, n
Define an array to hold n real numbers: Arr(n)
Input the number of elements in the array: n
While i <= n
    Input Arr(i)
    i ← i+1
End loop
i ← 1
While i <= n/2
    temp ← Arr(i)
    Arr(i) ← Arr(n – i + 1)
    Arr(n – i + 1) ← temp
    i ← i+1
End loop
i ← 1
While i <= n
    Print Arr(i)
    i ← i+1
End loop
Stop
```

Example20: Write the pseudocode to check whether a string is a palindrome or not.

```
Define variables: i ← 1, temp, len ← 0
Define two string variables: String1, String2
Input string to check: String1
String2 = String1
Repeat
    If String1(i) <> End of String
        len ← len+1: i ← i+1
Until End of String
i ← 1
While i < = len
    String2(len – i + 1) ← String1 (i)
    i ← i+1
End loop
If String2 = String1 then
        Print "The string is a palindrome"
Else
        Print "The string is not a palindrome"
Stop
```

The Fact File

- **The Program Definition Phase is used to work out what must be done to solve the problem and is used to extract from the problem statement the different inputs, outputs, rules etc. that must be incorporated in the program.**
- The Program Design Phase deals with the 'how' part of the programming. Since there can be more than one way to solve a problem it may not be always easy to find the best solution to a particular problem.
- **The Program Coding Phase is the actual coding of the program i.e. it is the process of translating the program logic into program code in a step by step manner using a particular programming language.**
- The Program Testing & Debugging Phase is used to check if the program runs correctly i.e. whether we get the desired outputs. For a good program, even if invalid inputs are supplied the program should be able to detect them.
- **The Program Documentation Phase is required to make a program code properly understandable. Comments should be inserted at strategic points to highlight the utility or use of a particular section.**
- An Algorithm is a sequence of precise and unambiguous instructions designed in such a way that if the instructions are executed in the specified sequence the desired result is obtained within a finite number of steps.
- **Sometimes by increasing the amount of space used for storing the data, the time required for processing the data may get reduced. On the other hand reducing the space for storing the data can lead to an increase in the time required to process the data. This is known as Space-Time Trade-off.**
- When an algorithm is expressed in a pictorial manner with special symbols to indicate the different types of instructions, then it is called a flowchart. The actual instructions are written inside boxes of different shapes indicating different functions. Solid lines with arrows indicating the flow of programs are then used to connect the boxes.
- **Pseudo means imitation and code means the instructions for the program. The pseudo or false instructions are written in a common language in a structural form that resembles computer instructions**

Review Questions

Q1. **Multiple Choice Questions. Select any one from the four options.** **1 each**

i) An algorithm:
   a. May or may not produce the exact solution to a problem
   b. Should get the result within a finite time limit
   c. May take any amount of time to process the data
   d. May not follow a definite sequence of steps

ii) When an algorithm is expressed in a pictorial manner with special symbols to indicate the different types of instructions, then it is called a:
   a. logic chart b. design chart c. flowchart d. algo chart

iii) What type of an operation is denoted by the Terminal symbol in a flowchart?
   a. START b. STOP c. HALT d. All of these

iv) What type of an operation is denoted by the Parallelogram symbol in a flowchart?
   a. calculation b. input or output c. decision d. process

v) What type of an operation is denoted by the Rectangle symbol in a flowchart?
   a. input or output b. terminal c. process d. decision

vi) What type of an operation is denoted by the Rhombus symbol in a flowchart?
   a. terminal b. decision c. input or output d. process

Q2. **Short Answer type questions:** **1 each**

i) What is an algorithm?
ii) Write the algorithm to increment a variable num by 1.
iii) What is a flowchart?
iv) Draw the flow chart to decrease the value of a variable num by 1.
v) State one advantage of using a flowchart.
vi) State one disadvantage of using a flowchart
vii) What is the use of the Parallelogram symbol in a flowchart?
viii) What is the use of the Rectangle symbol in a flowchart?

ix) What is the use of the Rhombus symbol in a flowchart?

x) What do you mean by a pseudocode?

xi) State one difference between a pseudocode and an algorithm.

xii) State one advantage of using pseudocode.

xiii) State one disadvantage of using pseudocode.

xiv) Write the pseudocode to increment a variable y by 1?

xv) Name any one of the basic logic structures that is used to write a pseudo code.



Q3. **Long Answer type questions:** **7 each**

i) Write the algorithm to input the length and breadth of a rectangle and calculate and display i area and perimeter. Explain any two phases of program development. 3+2+2

ii) Write the algorithm to input the number of terms for the following series and print the terms of th series up to that number: 5

1, 5, 25, 125, ...

State any two advantages of using a flowchart. 2

iii) Write the algorithm to input three numbers and print the larger of the three numbers. 5

Explain the terms 'sequential logic' and 'iterative logic'. 2

iv) Draw the flowchart to input two numbers x and y and print the result x^y without using any powe or exponent operator [hint: use iterative logic]. 5

Explain the terms 'conditional logic' and 'pseudocode'. 2

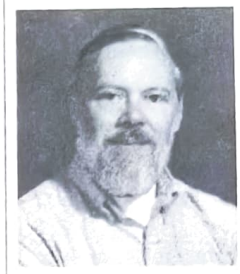
v) What is a flowchart? Draw the flowchart to find the sum of the numbers which are divisible by 7 i a set of N numbers input by the user. 2+5

CHAPTER 8
Programming in C: General Concepts

- **Introduction** 8-1
- **Structure and Components of a C Program** 8-1
- **Types of Data** 8-6
- **Constants and Variables** 8-8
- **Declaring Constants** 8-9
- **Declaring Variables** 8-11
- **Writing, Compiling, and Running a program in C** 8-12

8.1 Introduction

C is a programming language that was **developed at AT&T's Bell Laboratories in USA in 1972**. It was designed and written by **Dennis Ritchie** (picture on right) and it slowly replaced other contemporary languages like PL/I, Algol, Pascal, & APL. Since then C has become a popular programming language as it is reliable, simple and easy to use. In the early days of high level languages, the need for a **common language that could do all types of jobs** was felt and ALGOL60 was developed to meet the purpose. The series of programming languages that led to the development of C after ALGOL60 include – CPL, BCPL, B and finally Ritchie developed C by combining the features of B & BCPL along with his own ideas.



"C++ and Java, say, are presumably growing faster than plain C, but I bet C will still be around."
Dennis Ritchie
(1941 – 2011)

There are two types of programming languages:

1. Problem Oriented Languages or **High Level Languages** like Fortran, Basic, Pascal etc.
2. Machine Oriented Languages or **Low Level Languages** like Assembly Language, Machine Language

C is a language that stands in-between these two types and is often called a **Middle Level Language** as it was designed to have the positive points of both the types of languages. One of the unique features of the C programming language is the **way it handles the use of available memory**. Unlike other programming languages, C gives the user the freedom to place the variables in particular memory locations. Taking advantage of this feature the programmer can write faster executable code and remove run-time errors very easily. However the **source code of C needs to be compiled** using a C compiler before it can be used.

8.2 Structure and Components of a C Program

Every C program consists of **one or more modules** called **functions**, which are a series of instructions to the computer to **perform a specific task**. Of these functions, the **most important function is the main() function**. Every executable C program should contain the **main()** function, because the execution of the C program starts from the **main()** function and terminates at the **main()** function. All other functions or modules are called from the **main()** function or from the other functions.

The main( ) function

Many **functions** that we will be using are **already written and compiled** and are available in the function libraries supplied with the compiler. These are known as **library functions**. Thus instead of writing all individual instructions, one just tells the compiler to use one of its standard library functions wherever required. Only when one wants to perform a task that is not in the function library, one has to **write his own function**. Such a function is known as a **user defined function**.

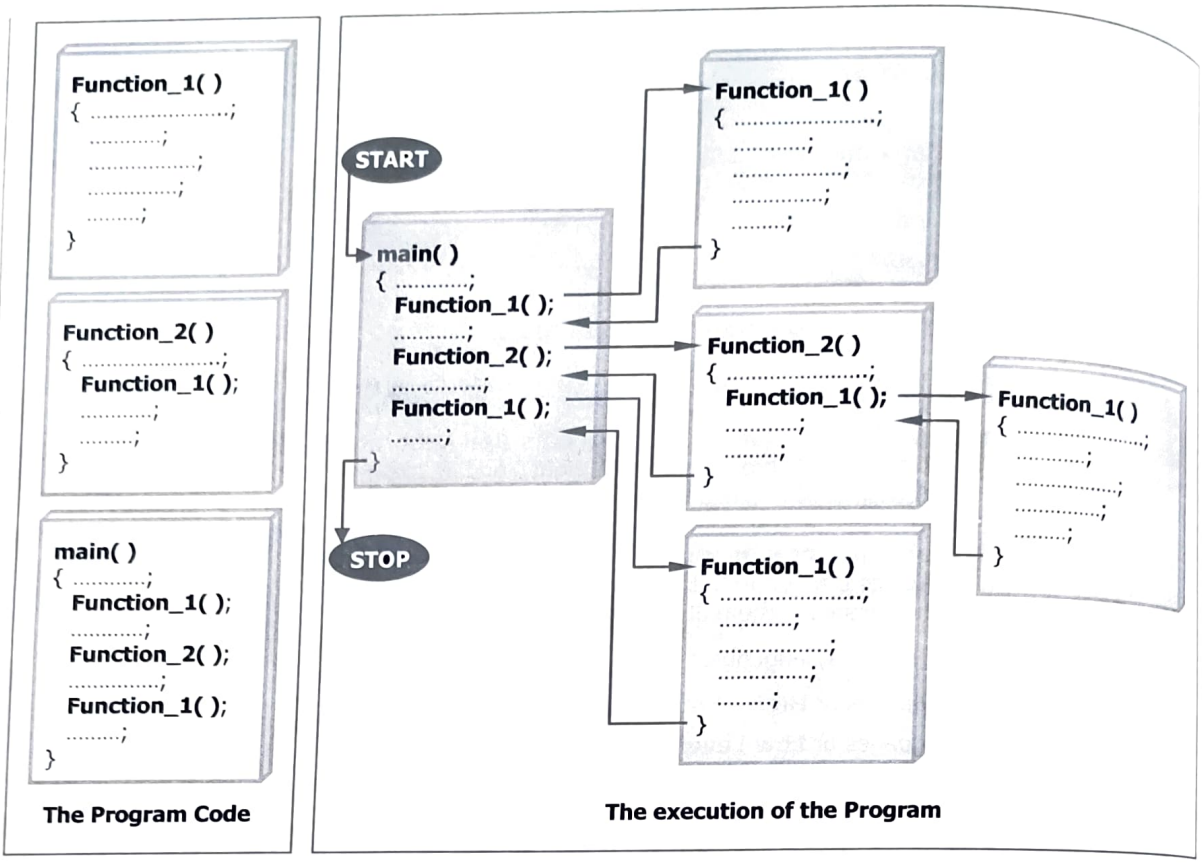
For example though C does not have any inbuilt command to display characters on the screen it can use a function called **printf()** to do the same job. Hence, one does not have to always write the code each time one wants to display something on the screen but will have to only call the function **printf()** from the **function library** to do the job. Whereas suppose if someone wants to find the maximum between two numbers using a function he has to write his **custom function** and save it for future use. Whenever required, he will simply have to call the function created by him and it will perform the required function.

The C language was developed to write the code for the UNIX operating system.

The diagram in the next page shows the general format of a C program with three modules/functions viz. **main()**, **Function_1()** and **Function_2()**. The right hand diagram shows the relationship between the different functions and the flow of logic. As can be seen, the program starts from the first line of the **main()** function. The **main()** function in turn calls **Function_1()** and **Function_2()**. **Function_2()** in turn again calls **Function_1()**. At the end of execution, each function can return a value to the calling function. Finally the program stops at the end of the **main()** function. We will get a better understanding of the structure as we write actual programs.

The modern programming language Python was written using C.

Relation between different modules in a C program



C has its own set of reserved words or Keywords with specific meanings that are used to construct a C program.

Components of a C Program

• Components of a C Program

Like the letters in the English language that are used to form words and sentences, the character set of C that are used to write a C program also consists of alphabets, digits and special symbols. These include:

- **The Alphabets**: A, B, C, ..., Z, a, b, c, ..., z
- **The Digits**: 0, 1, 2, ..., 9
- **Special Symbols**: + - * / % # ! = { } [ ] ( ) , ; : ? & ^ | < > .

In general a **C** program consists of the following **building blocks** or **tokens** as discussed below:

Keywords

a. **Key Words**: Like the words in a particular language that are used to write the sentences, C has its own set of reserved words or **keywords that are used to construct a C program**. The original C language supports **32 basic keywords**. These keywords cannot be used for any other purpose.

auto	*break*	*case*	*char*	*const*	*continue*	*default*	*do*	*double*	*else*	*enum*
extern	*float*	*far*	*for*	*goto*	*if*	*int*	*long*	*near*	*register*	*return*
short	*signed*	*static*	*struct*	*switch*	*unsigned*	*union*	*typedef*	*void*	*while*	

Identifiers

b. **Identifiers**: Identifiers are **names that are used to identify the different parts of a program**. These names can be given to things like variables, constants, functions, arrays, structures etc. that are used as the building blocks of a program (don't worry, you will get to know all these parts in due course). The name of an identifier can be anything but should follow certain naming rules as stated below:

- It **cannot be any keyword** or reserved word in C
- It can **contain all the upper and lowercase alphabets** ('a' to 'z' and 'A' to 'Z') and the **underscore** (_) character
- It can contain **all the digits** from 0 to 9, but **cannot start** with a digit
- It cannot contain a blank space

Note that **identifier names are case sensitive** i.e. **NAME** and **name** are different identifiers in C. We now give some examples of valid and invalid identifier names.

Identifiers are names that are used to identify different parts of a program like variables, literals, functions etc.

Valid Identifier Names		Invalid Identifier Names	
		Name	Reason
PI	CONST_1	007	All digits
rate	RateOfInterest	29TH_YEAR	Starts with a digit
average	_Account_No_1	DATE-OF-BIRTH	Contains hyphen
num2	DATE_OF_BIRTH	area of circle	Contains space
factorial	_20TH_AVENUE	"ACCOUNT NUMBER"	Contains double quotes
table25	_double	double	Keyword

Valid and Invalid Identifier names

c. **Literals**: Apart from the instructions a program may also contain some fixed values which are used as part of the data processing. These values can be both numeric (like the value of pi as 3.14159) and alphanumeric (like an input prompt as "Enter your name:") in nature. Such **values that do not change in a program** and remains same independent of the number of times are program is run are called literals. These are sometimes called **constant values** also depending upon their use.

Literals

d. **Operators**: Certain special symbols or combination of symbols are used in a C program to do **specific operations** like adding two numbers, comparing two values, incrementing a number etc. These symbols are called operators and normally require one or more operands or values to work upon. These include:

& + - * / % = > < >= <= == != ++ -- += -= *= /= %= -> && || . etc.

Operators

e. **Punctuators**: Certain symbols are used in a C program as punctuations, similar to the punctuations we use in an English sentence. These include: ( ) { } [ ] ; : , etc.

Punctuators

f. **Instructions**: Using the above components meaningful instructions can be constructed in C, which when run properly can solve a particular problem. There are **four basic types of instructions** in C. These are:

- **Type declaration instructions**: These are used to declare the type of variables in a C program.
  Example: `int x, y=0;`
- **Input/Output instructions**: These are used to enter a data for calculation and to get the output result i.e. for doing input and output operations.
  Example: `scanf("%d", &x); printf("\nThe entered value = %d", x);`
- **Arithmetic instructions**: These are used for doing calculations involving variables and constants.
  Example: `y = y + a*b + 2.5;`
- **Control instructions**: These are used for controlling the flow of a program i.e. to control the execution of various statements in a C program.
  Example: `If(x%2==0) printf("\nEven Number"); else printf("\nOdd Number");`

The different building blocks of a program are also called Tokens

g. **Functions**: A function is a **group of statements that perform some meaningful work** and normally returns a result back to the point from where it was used. You can supply data to a function for processing and get the result back from it. You can have both user defined and library functions. The process of defining a function is called 'function definition' and the process of using it is called a 'function call'.

Functions

Now let us come to the **structure of an actual C program**. In writing a C program as per convention, the following rules should be followed:

- **Lowercase alphabets** should be used for keywords. Function names and variable's names should preferably be in lowercase, whereas capital letters are used to indicate constants.
- **Blank spaces** should be put between two words, instructions or wherever required to improve readability.
- C is often called a **free form language** as there are no specific rules to position a statement. However **proper indentations** in writing a program are encouraged to improve readability and help in debugging.

So the basic structure of a C program looks like this:

```
inclusion_of_header_files          ⇒ Pre-processor directives are written at the beginning
main()                             ⇒ Indicates this to be the first function to start when the program runs
{                                  ⇒ The main function starts here with a delimiter ' { '
   statement_1;                    ⇒ Each Statement is terminated by a semicolon ' ; '
   statement_2;
   . . .                           ⇒ These are the instructions to be performed
   Statement_n;
}                                  ⇒ The main function ends here with a delimiter ' } '
```

Structure of a C Program

The function of the pre-processor directives will be discussed later. Presently let us write a simple program in C to input a number, find its square and print the result.

Example of a C program

Example1: To find the square of a number input by the user.

```
1  /*Program-01: To find Area of a Circle*/
2  #include <stdio.h>
3  int main()
4  {
5   float radius, area;
6   printf("Enter radius of circle: ");
7   scanf("%f", &radius);
8   area = 3.1416*radius*radius;
9   printf("\nArea = %f", area);
10  return 0;
11 }
```

⇒ Comment line
⇒ **Pre-processor directive** to include the stdio.h file
⇒ The **main** function header
⇒ Delimiter for beginning of body of main( ) function
⇒ Declaration of variable with **variable type** as `float`
⇒ Input asked for radius of circle
⇒ Reads the input and stores it in the variable `radius`
⇒ The area is calculated and stored in the variable `area`
⇒ The calculated area is printed on the screen
⇒ Value 0 is returned to the Operating System at the end

A **Comment** is a piece of information written (but not run) for a better understanding of the program.

Let us analyse the program line by line to get a **preliminary idea about an actual C program**. It does not matter if you cannot understand every bit of the code now. The **line numbers** at the beginning of each statement are used to **help analyse** the program. **An actual C code should not have any line number**.

Line-1

Using Comment

Comment: For a **better understanding of the program** logic, sometimes it may be helpful to **add comments** in a program. The program name, programmer name, the purpose of the program, version and any other relevant information may be included in a program. Such **information** which is **not a part of the program coding** can be put inside the program by writing them between the symbols **/*** and ***/**. Whenever the compiler encounters the symbol combination **/*** and ***/** it **ignores whatever is written within these**.

Line-2

#include directive

The #include directive: As discussed earlier, the standard library provides several pre-written library functions, data type definitions, constant definitions, and macros that can be used in your program. However **to use these utilities certain specific information needs to be included** within the main portion of the program (you will learn more about these in later chapters). These information are generally stored in **specific files** which are supplied with the compiler and can be accessed using the **#include** command. The files that need to be included are called **header files** and usually have a "**.h**" extension. **Header files** need to be included at the top or head portion of a program before the **main()** function.

Header files

The **header file** that we will use in almost all C programs is the `stdio.h` file (the initials *stdio* stand for "**st**and**a**rd **i**nput **o**utput"). This header file contains all the information that the compiler needs for functions dealing with input and output operations and working with the disk, monitor and printer. The command to include this header file will look like: `#include<stdio.h>`

Pre-processor directives are carried out before the actual compilation process starts.

This command is called a **pre-processor directive** that tells the compiler to use the information in the header file called `stdio.h` and **insert the contents** of the file into the program before the beginning of the processing/compilation (i.e. conversion of the source code to machine code) process.

Surrounding the name of the header file with the symbols **<...>** (also called angled brackets) tells the compiler that the file may be located in the **default "include folder"**. This is the folder where the compiler's installation program places all the header files.

stdio.h is a header file that is used for standard input and output functions.

There is another way of writing the above pre-processor directive: `#include "stdio.h"`. Surrounding the file name `stdio.h` by **double quotes** tells the compiler to look for the file first in the **current directory** i.e. the directory where the user has been working currently. If the file is not available in the current directory then the compiler will search for the file in the default directory.

Line-3

The main() function header

The main() function header: **The most important function** or module **in C** is the **main()** function. A function definition has two parts – the **header** and the **body**. The header gives information about the function return data type (`int` in this case), the function name (`main` in this case) and the function argument list (represented by the pair of brackets `()` after the name). You will know about these in detail when we discuss about user defined functions in general in a later chapter.

For the time being just write this line when you are defining the `main()` function for your program. Remember that **every executable C program should contain the `main()` function**, because the **semicolons after a function header like `main()`**.

Line-4 & Line-11

Delimiters: After the function header line comes the body of the function. It contains the code that is executed as per the function specification. The code inside the body of the function is enclosed within a pair of opening "`{`" and closing "`}`" braces. The opening and closing braces are called delimiters and they mark the beginning and end of a block of code. Whatever work is to be done by the `main()` function is to be written within the two braces (or curly brackets).

Delimiters { }

Line-5

Variable declaration: In this line, the variables or memory spaces required to store the values used in the program are declared. **Variables** serve **as containers** for storing different data that are used during a program execution. Here we have declared two variables `radius` and `area`. The first one is used for storing the input radius of a circle and the second one is used for storing the result of finding the area of the circle.

Variable declaration

Along with the variable names you also need to declare the type of the data. The type is to be put before the variable names. This helps C to allocate specific number of bytes for each data stored. In this example we have declared the data type as `float` for both the variables. Here `float` stands for real or floating point numbers i.e. numbers having a fractional or decimal part.

Statement Terminator: Each instruction in a C program must end with a **semicolon** "**;**" which is also known as a statement terminator. The semicolon tells the compiler that it has reached the end of the instruction and what comes next will be another instruction or the end of the program. However this does not imply that every line must end with a semicolon rather **every instruction should end with a semicolon** and an instruction can consist of more than one line of code on the screen.

Statement Terminators

Line-6 & Line-9

Output Instruction: These statements are used to display something on the screen. These use the library function `printf()` to do the job. In line-6 the `printf()` function is used to display the fixed string or literal "`Enter radius of circle: `" on the screen. Whereas the second `printf()` statement in line-9 is used to display the area of a circle with the input radius. More of this will be discussed in later chapters.

Output Instruction

Line-7

Input Instruction: This statement is used to input something from the keyboard. It uses the library function `scanf()` to do the job. The value entered by the user from the keyboard is stored into the variable `radius`. More of this will be discussed in later sections.

Input Instruction

Line-8

Arithmetic Instruction: The area of the circle is calculated in this statement. The result is then assigned to the variable `area`, where it is stored for future use. In this case the result is displayed in line-9 using the `printf()` statement. Remember that all calculations should be placed on the right hand side of the equal sign i.e. the result of a calculation is assigned from the right of the equal symbol to the left variable.

Arithmetic Instructions

Line-10

The return statement: When a computer finishes performing the instructions, the program stops and the computer returns to the state it was in before the program had started. In general the return of control to the system takes place automatically, but for some compilers the **return** statement needs to be inserted just before the closing braces of the **main()** function. The '0' (zero) after the return keyword indicates to the operating system that the program has terminated successfully. There are other functions of the return statement that will be discussed in later sections. Note that the number '0' returned by `main` is an integer (or whole number) and is related to the keyword `int` written at the beginning of the function header in line 3.

The return statement

In the above examples the combination `\n` has occurred with the `printf()` statement of line-9. It is known as an **escape sequence** and it is used to start a new line. Whatever is written after this is displayed on the screen from a new line. Escape sequences will be discussed in detail at a later stage.

If you cannot understand every detail of the above program there is nothing to worry, for you have just begun and it is natural that you may find some (if not all) concepts a little difficult. In the subsequent sections there are ample opportunities to clarify your doubts and improve understanding.

8.3 Types of Data

Data types

The basic data types in C include int, float, double, and char

Information given to a computer is generally known as data. The computer processes this data and gives information as output. But before supplying data to a computer it must be told what type of data it is dealing with so that it can set aside sufficient memory space to store each data item **without unnecessarily wasting any memory**. (Note that in earlier days of computing memory was really costly and one of the aims of a good program was to minimise the requirement of memory).

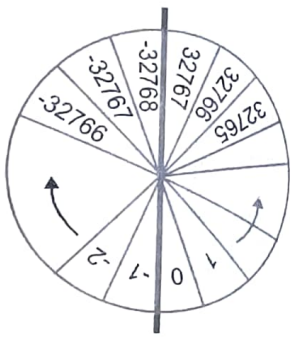
We specify a **data item** by the **type of data** it holds. In examples 1 and 2 we have used the terms `int` & `float` to specify that the data that will be input from the keyboard will be of `int` type and `float` type. We now state the different types of data that are supported by C. There are **4 primary types** of data. These are `int`, `float`, `double`, and `char`. Each one is described below.

Integer type data

An integer type data represents a whole number and is at least 2 bytes in size with a range of at least -32768 to +32767

- Integer type data (2 bytes long), represented as int

Purpose: An integer number is a number with **no decimal point** i.e. a **whole number**. It can be a **positive** number or a **negative** number or **zero**. It is used generally for **counting purposes** or for calculations **involving only whole numbers**.



Byte Requirement: As per the original standard, each piece of integer data requires **2 bytes of memory space** i.e. 16 bits. Hence the total number of values possible is equal to 2^{16}=**65536**. It is represented by **keyword `int`**.

Range: A 2 byte integer variable can store values **from –32768 to 32767 including 0**. For signed numbers, you can think of the numbers as placed on a number wheel as shown in the diagram above. The wheel is divided into 65536 sectors corresponding to the 2^{16} possible values of an integer.

To **accommodate both positive and negative numbers** evenly, the wheel is divided into **two sections** with each section containing 65536/2 = 32768 values. The right hand section contains values from 0 to 32767 (total 32768 values, including 0) and the left hand section contains values from -1 to -32768 (total 32768 values). Thus if we traverse the wheel clockwise, the position after 32767 is not 32768 but is –32768. Similarly (–32768 – 1) is not equal to –32769 but 32767.

Floating point type data

An float type data represents a real or decimal number and is at least 4 bytes in size with a range of at least -3.4×10^{-38} to $+3.4 \times 10^{+38}$

- Floating point type data (4 bytes long), represented as float:

Purpose: To **represent real numbers** i.e. numbers having **both an integer part and a fractional part**, a computer typically stores the real number as a combination of three parameters viz. the sign, mantissa, and an exponent part (for details refer to the Data Representation chapter). These are the computer analogues of scientific notation used in mathematics and are called **floating-point representations**.

Simple floating-point numbers are 2.54, 6.023, 13.6 etc. However floating-point numbers can be extremely large or small and hence are expressed as exponential numbers. For example the number 2.9E+19 is read as 2.9 with an exponent of +19 and indicates that move the decimal point to the right by 19 places by adding adequate numbers of zeroes to get the actual number. In algebraic notation the number is represented as 2.9×10^{19}. Thus the above number is actually equal to: 29,000,000,000,000,000,000. Similarly a fractional number can be expressed as 5.235E–8, where the negative exponent states that one should shift the decimal point to the left by 8 places to get the actual number. Thus the actual number is equal to: 0.00000005235 and its algebraic equivalent is 5.235×10^{-8}. Accordingly the Avogadro number is represented as 6.023E+23 or as 6.023E23.

Byte Requirement: A **floating-point** number is **4 bytes long** and is represented by the keyword `float`.

Range: The range of a float is from **–3.4E–38 to +3.4E+38**. As floating-point variables make an approximation, there is limit to the precision of floating point numbers. For example though the number 5.21342345671215 is well within the range of a `float` but it may be stored as 5.213423. A `float` is a **single precision number**, which means that the precision of a float is **limited up to 7 decimal places**.

- **Double type data (8 bytes long), represented as double:**

Purpose: This data type is also used to represent real numbers i.e. numbers having both an integer part and a fractional part. This is called a **double precision floating point number representation** and is used for variables that require greater precision. Unlike a floating point number a double type variable can be **accurate up to 15 digits after the decimal**.

Byte Requirement: A **double** type variable is **8 bytes long** and is represented by the keyword `double`.

Range: The range of a `double` is from –1.7E–308 to +1.7E+308.

Double type data

An **double** type data represents a real or decimal number and is at least 8 bytes in size with a range of at least -1.7×10^{-308} to $+1.7 \times 10^{+308}$

- **Character type data (1 byte long), represented as char:**

Purpose: This type of data represents a **single letter**, numeral or **other keyboard characters**. Character type data can be used where the input required is in the form of a single character (like selecting a choice number or alphabet from a set of multiple choices).

Byte Requirement: For each piece of character type data the computer reserves just **one byte of space**. Each `char` type data has an equivalent integer representation varying from 0 to 255.

Range: Characters thus include the 26 uppercase and 26 lowercase alphabets, the 10 numeric digits and other punctuations and symbols that can be entered from the keyboard. Note that a **char** type data can also store **integer type data**. We give below the different characters available as `char` type data:

Uppercase alphabets	A, B, C, D, E,..... , X, Y, Z
Lowercase alphabets	a, b, c, d,..... , x, y, z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special symbols	~ ! @ # $ % ^ & * ( ) _ + = - | { } [ ] \ " : ' ; < > , . ? /

An **char** type data represents a character and is 1 byte in size.

String type Data: A string is usually a **group of characters** such as word, phrase or sentence. **C does not have a string data type** but treats strings as **character arrays**. More about strings will be discussed later.

- **Using data modifiers:**

Using data modifiers

You can **modify the range of these primary data types** using a set of keywords called **data modifiers**. They define the different amounts of storage that are possible for a variable of a given data type. The amount of storage allocated is however not absolute. ANSI has the following rules in this regard:

- short int <= int <= long int
- float <= double <= long double

Let us now discuss about the different modifiers used in C to declare a variable named `num`.

Data Type	Meaning
`signed char num;`	Of the same size as char, but guaranteed to be signed
`unsigned char num;`	Of the same size as char, but guaranteed to be unsigned

`Short num;`
`short int num;`
`signed short num;`
`signed short int num;`
	All these declarations are used to indicate a short signed integer type data. It is capable of containing values which are in the range –32768 to +32767 (i.e. both negative and positive numbers)

`unsigned short num;` `unsigned short int num;`	The same as short, but unsigned i.e. will store only positive values in the range 0 to +65535
`int num;` `signed int num;`	Basic signed integer and is capable of containing values which are at least in the range –32768 to +32767
`unsigned num;` `unsigned int num;`	The same as normal integer type data, but unsigned i.e. capable of containing values which are at least in the range 0 to +65,535

`long num;`
`long int num;`
`signed long num;`
`signed long int num;`
	Signed integer value and is capable of containing values which are at least in the range –2,147,483,648 to +2,147,483,647

`unsigned long num;` `unsigned long int num;`	The same as long type data, but unsigned i.e. capable of containing values which are at least in the range 0 to +4,294,967,295

Meaning of different data modifiers

Modifiers are used to change the range of a given data type.

Data Type	Meaning
`long double`	Real floating-point type data with an extended precision of 19 decimal places. It is at least 10 bytes in size and the range of a long double is from –3.4E–4932 to +1.1E+4932
`signed char num;` `unsigned char num;`	Character type data are inherently of type int (as these basically store the ASCII value of a character). Accordingly a signed char has range of -128 to +127 and an unsigned char has a range from 0 to 255

8.4 Constants and Variables

Constants and Variables

All **working data in a program is stored in the computer memory** i.e. RAM. The RAM is not a small space and every location or byte in the RAM has a specific address. Data can be stored in or accessed from a given location in the RAM by using this specific address.

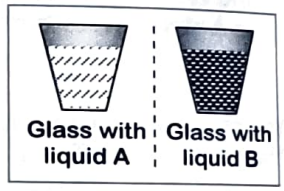
However, it is not convenient to access data by using the memory address where it is stored. A better **alternative is to use a name** to identify the specific memory address location. Just as it is easier to identify a place like Howrah Station by its name, rather than by its specific postal address, it is easier to identify a memory location by an assigned name than by its actual memory address.

Variables

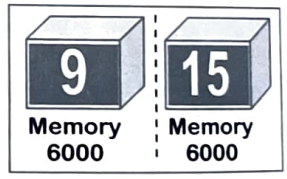
A Variable is a temporary storage location for data that can change in a program.

Variable: A **variable is a temporary storage location for data that can change in a program**. These can be used to store data input by the user or store the result of a calculation in the program. Variables let you **assign a meaningful name** to a memory address for stored data in your program. Note that there can be more than one variable in a program. The operating system keeps track of the **identifier name of a variable** associated with a given memory address. When you try to access a variable using its name, the operating system gets the address from the corresponding name and gets the data from the actual address.

One can **think of variables as containers** that can be used to **hold some values** used in the program. As you can use the same container for storing various items, similarly a given variable can be used to store different values. However, **only one value can be stored at a time in a given variable**. When you store a new value the old value gets overwritten and is lost.



For program-01, every time you run the program you can enter a different value for the `radius` and accordingly get a different value for the `area`. The values in `radius` and `area` change every time i.e. vary and hence are called variables.



Constants

A Constant is a storage location for data that never changes in a program.

Constant: Sometimes a **value in a program may never change**. It remains the same no matter how many times you run it. Such a value can be **stored as a constant**. Once a value is stored as a constant, it cannot be changed when you run the program. You need to change the code to change the constant value. Constants are also called '**literals**'. For example the value of π i.e. 3.14159 (approx.) is a constant. **Constants** are defined when certain **fixed data are being used several times** in the program.

For example if the value of π is used several times in a program then it is better to define a constant called **PI** and store the value in the constant. In this way one will not have to every-time write **3.1415** wherever the value of π is needed but will simply have to put PI. This **saves time and chances of errors**. Moreover if later someone wants to increase the accuracy to 3.141592 then he/she will have to make the change only at one point where the constant was defined. This **saves time** and **possible chances of any mistake**.

Some basic Constants

The **basic types** of constants and the rules for constructing the different constants are given below:

Type	Rules
Integer	• It must have **at least one digit** and **must not** have a **decimal point** • It could be either **positive** or **negative** (no sign indicates a positive number) • **No** comma or blank should be present within an integer constant • The allowable range of an integer constant is from –32768 to +32767 including 0 for a 2 byte int
Octal Integer	• It must have **at least one digit** and it **must have** a **0** (zero) before the number • **Allowed digits** are 0, 1, 3, 3, 4, 5, 6, and 7 • **No** comma or blank should be present within an octal integer constant
Hexadecimal Integer	• It must have **at least one digit** and it **must have** a **0X** (zero x) or **0x** before the number • **Allowed digits** are 0, 1, 3, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F • **No** comma or blank should be present within a hexadecimal integer constant

Type	Rules
Floating point Number	• It must have **at least one digit** and **must have** a **decimal point** • It could be either **positive** or **negative** (no sign indicates a positive number) • **No comma or blank** should be present within an integer constant • It can be written in either a **fractional form** or as an **exponential form** • When expressed in exponential form, ○ The **mantissa** part and the **exponential** part should be separated by the letter **e** or **E** ○ The mantissa part may have a **positive** or **negative** sign ○ The **exponent** must have at least one **+ve** or **–ve** integer (default is positive) • **Range** of real constants can be from **`–3.4e–38 to 3.4e38`**
Character	• It can be either a **single alphabet**, single **digit**, or a single **special symbol** • Each constant should be enclosed within **single quotes** like **'A'** • The maximum length of a character constant is one character
Strings	• It can be **any combination of characters** • Each constant should be enclosed within **double quotes** like "This is a String"

Given below is a **chart of valid and invalid constants** of the different data types, constructed by taking into consideration the rules that have been discussed above.

Integer		Octal Integer		Hex Integer		Float		Character	
Valid	**Invalid**	**Valid**	**Invalid**	**Valid**	**Invalid**	**Valid**	**Invalid**	**Valid**	**Invalid**
32757	32,757	**076**	29	**0X2A5**	5,201.32	**+56.23**	5	**'A'**	Hi
5	32.057	**02**	-0.2e+2	**0X5**	0.0	**0.0**	5,201.32	**'z'**	A
0	0.0	**0**	0.0	**0x0**	010	**2.0**	0	**'$'**	23
-10	6E-9		019	**0X**	0F	**-3.2e-5**		**'?'**	2e5
-258	2E5				-0.9E+6	**-0.2E+2**		**'4'**	'char'

Different valid and invalid constant values

Constants and variables can again be classified as Primary and Secondary based on their type.

Primary Constants/Variables: In case a data is of a **basic data type** i.e. **`int`**, **`float`**, **`double`** or **`char`** then it is called a primary constant or variable.

Primary and Secondary data

Secondary Constants/Variables: Apart from the basic data types, data can also be of type array, pointer, structure, union or enum. Such a data type is called a secondary constant or variable.

8.5 Declaring Constants

Declaring Constants

Now let us find out how to declare a constant or a variable i.e. how to let C know which data is a constant and which one is a variable.

Declaring a constant (also called **symbolic constant**) means telling the C compiler the **constant's name and value and an indication that the value will not change**. The rules for naming a constant are same as that of naming identifiers. All constants must be declared before they can appear in an executable statement.

A constant can be **declared in two different ways**. The first one uses the **pre-processor directive `#define`** to define a constant and the second one uses the **`const` keyword**. There is a difference between these two ways as discussed later.

• Using the #define directive to use a constant:

Using #define

The **`#define`** directive creates **symbolic constants** i.e. constants represented as symbols and macros (to be discussed later). While using a symbolic constant the constant name should appear at the places where the value (numerical or string) indicated by the constant is required. During compilation, **before the compiler starts to create the object code, it simply replaces each occurrence of the constant name with its value**. A symbolic constant has to be defined **before** the **main()** function. The syntax of declaring a symbolic constant is:

the numeric value, character or string that the constant represents

```
#define NAME value
```

symbolic name to represent the constant, typically written in uppercase

A `#define` directive creates a symbolic constant in C, that is also called a macro.

Note that a symbolic constant definition **does not end with a semicolon**, as this is not a true C statement.

Let us write a **modified version of program-1** using the **#define directive** to declare the constant **PI**.

Program using #define

```
1  /*Program-01a: Using constant in a program*/
2  #include <stdio.h>
3  #define PI 3.1416                          ⇒ Declaration of constant called PI using #define
4  int main()
5  {float radius, area;
6   printf("Enter radius of circle:");
7   scanf("%f", &radius);
8   area = PI*radius*radius;                  ⇒ Using the constant PI in the calculation
9   printf("\nArea = %f", area);
10  return 0;
11 }
```

In line-3 the statement `#define PI 3.1416` creates a symbolic constant `PI` and associates it with the value `3.1416`. During compiling the program wherever the compiler sees `PI` in the source code, it substitutes the value `3.1416` there. Therefore **after pre-processing of the source code**, line-8 will look like:

```
8  area = 3.1416*radius*radius;
```

There are two equal (=) symbols used with the #define directive to assign a value to the name.

Also note that **C does not have any operator to do exponentiation** i.e. radius^2 (in BASIC the operator ^ is used to get the power as radius^2). Thus radius is multiplied twice to get the result of radius^2. Powering can also be done using a loop or a library function, as will be discussed in a later chapter.

Examples of the use of **#define** for defining various constants are given below:

```
#define PI 3.1516                    ⇒ defining a float type constant
#define RATE 12                      ⇒ defining a int type constant
#define NUMBER 0.5                   ⇒ defining a float type constant
#define OPTION1 'A'                  ⇒ defining a char type constant
#define REPLY1 "Yes"                 ⇒ defining a string type constant
#define ASK "Press Y to Continue"    ⇒ defining a string type constant
```

C automatically assigns a data type based on the value given in the #define directive

The data type need not be stated explicitly when defining a constant. 'C' automatically assigns a data type based on the value given in the **#define** directive. Thus:

```
#define RATE 12      ⇒ will be assigned an int data type because 12 is an integer
#define PI 3.1516    ⇒ will be assigned a float data type as 3.1516 is a float
#define REPLY1 'A'   ⇒ will be assigned a char data type as 'A' is a character type data
```

Using const keyword

- Using the const keyword to use a constant:

The **const keyword** can be used in a program in the declaration of any data that you do not want to get changed. Therefore any `const` type data must have an initial value assigned to it as it is not easy to assign any value to a **const** variable later in a program.

```
const long double PI = 3.141592653589793238L;
```

Here the **letter L appended at the end of the value** indicates the value to be a `long double`. In the absence of the letter **L**, the value would have been taken simply as a `double` with up to 15 digits of precision. This is because the **default data type of a real number is** `double` and without an identifier at the end of the value, all real numbers are treated as a `double` with 15 places of precision.

Let us now again **modify program-1** using the **const** keyword to declare the constant π.

Program using const keyword

```
1  /*Program-01b: Using const keyword in a program*/
2  #include <stdio.h>
3  int main()
4  {const long double pi = 3.141592653589793238L;   ⇒ Declaration of constant called pi, using const
5   float radius, area;
6   printf("Enter radius of circle:");
```

```
7   scanf("%f", &radius);
8   area = pi*radius*radius;
9   printf("\nArea = %f", area);
10  return(0);
11  }
```

⇒ Using the constant pi in the calculation

In line-4 of the program shown above, a constant type **long double** data called **pi** has been defined using the keyword **const**. This means that the value stored in the memory location identified by the name **pi** cannot be changed normally. The value is used in line-8 to calculate the area.

The **main difference between the two ways of declaring a constant** is that in **Program-01a**, during the pre-processing stage, wherever the name `PI` is encountered it is replaced by the value `3.1416` and then the source code is compiled to form an executable file. When the program is run, no space is reserved in the data area of the memory for storing the value `3.1416`.

Whereas in **Program-01b** when the program is run, space is reserved in the memory for the constant `pi`. The value `3.14159265358979323 8L` is stored in the memory location reserved for `pi`. Whenever the value is required, it is fetched from the respective memory location and used in the program.

When a program is run, no space is reserved in the data area of the memory for storing any value defined using #define directive.

8.6 Declaring Variables

Just like constants, **variables also need to be declared** so that C can assign a portion of the memory to store the variable. By declaring a variable and giving it a name, C reserves memory space (e.g.: 2-bytes for int, 4-bytes for float etc.) to store the contents of the variable. The variable **should be defined before the program accesses it**. Thereafter the data item can be accessed simply by referring to the variable name.

Declaring variables

A **declaration** of a variable consists of a **data type followed by one or more variable names** separated by commas and the statement ending with a semicolon. **Variable names** are also **formed** following the rules for naming identifiers as discussed earlier.

variable names separated by a "," & ending with a semicolon

```
type name1, name2, name3;
```

variable data type

A declaration of a variable consists of a data type followed by one or more variable names separated by commas and ending with a ;

The following examples show how different variables can be declared.

```
int count, days;
```
⇒ 2 integer type variables called `count` and `days`

```
float area, rate;
```
⇒ 2 floating point type variables called `area` and `rate`

```
char initial;
```
⇒ 1 character type variable called `initial`

- **Assigning initial values to variables:**

Assigning initial values to variables

A **literal** is any piece of **data that one directly types in into a program**. It can be any number, character or string that one may **use as an initial value** in the program. Though a constant literal remains the same throughout the program but a literal assigned to a variable may change in course of a program. The following examples show different literals in a program.

```
#define PI 3.1415
```
⇒ Here 3.1415 is a constant floating point literal

```
const int factor=32;
```
⇒ Here 32 is a constant integer literal

```
int sum=1;
```
⇒ Here 1 is an integer variable literal

```
float rate=12.5;
```
⇒ Here 12.5 is a floating point variable literal

```
printf("Welcome to C");
```
⇒ Here "Welcome to C" is a string literal

A Literal is any piece of data that one directly types in into a program

Variables may have an initial value i.e. one may want a variable to have a certain value at the start of a program, though this value can change as the program runs. For example to find the sum of the terms of a series the variable sum needs to be initialised to 0 (zero). This **initial value can be assigned** either **as a declaration when the variable type is declared** or **as a separate instruction**. Thus:

```
int count=5, sum=0;
```
⇒ Here the variables `count` and `sum` are initialised during declaration

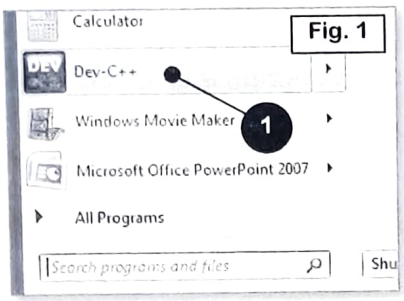
```
int count, sum;
count = 5;
sum = 0;
```
⇒ Here the variables `count` and `sum` are declared first and initialised later

Note: a `char` type literal should be put between two single quotes ' ' as, `char x = 'Y'`.

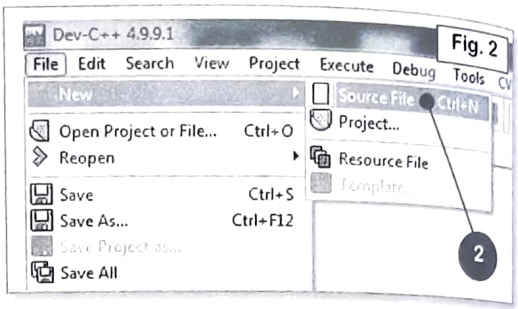
8.7 Writing, Compiling, and Running a program in C

Let us now see how we can actually write and run a C program. You can use various Integrated Development Environments (IDEs) like Turbo-C, Code blocks, Dev C/C++ etc. for writing, compiling, and running a program. All these are freely available and can be downloaded from their respective websites using a search. We will be using Dev C/C++ for our book which can be downloaded free from the **sourceforge** website **http://sourceforge.net/projects/orwelldevcpp/files/latest/download**

Install Dev C++ using the setup file. An icon for Dev-C++ will appear in the **Start Menu** as shown in the figure-1.

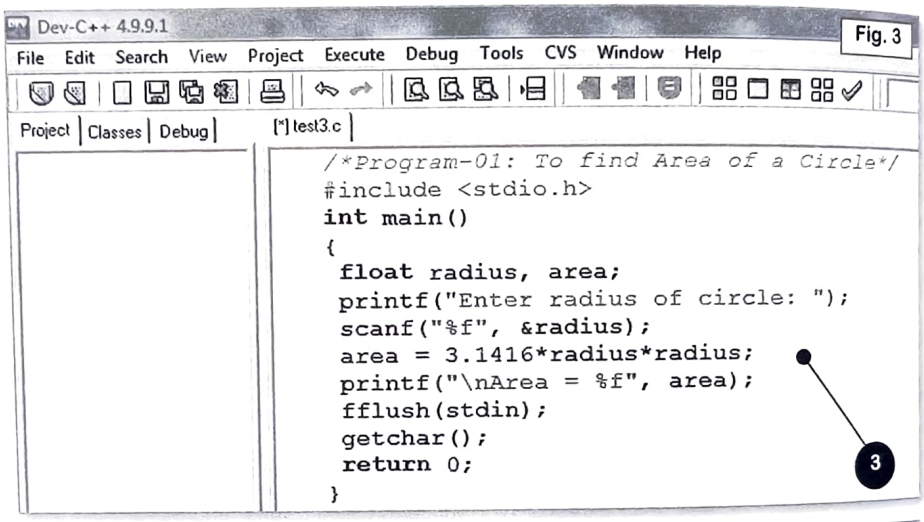


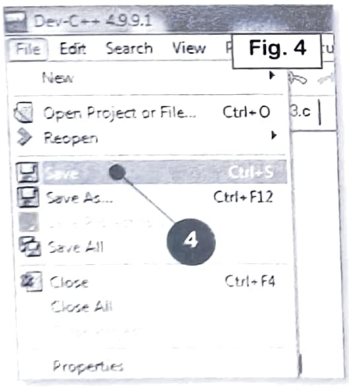
Click on the icon (or double click on the shortcut icon on the Desktop) to open the Dev C++ IDE.



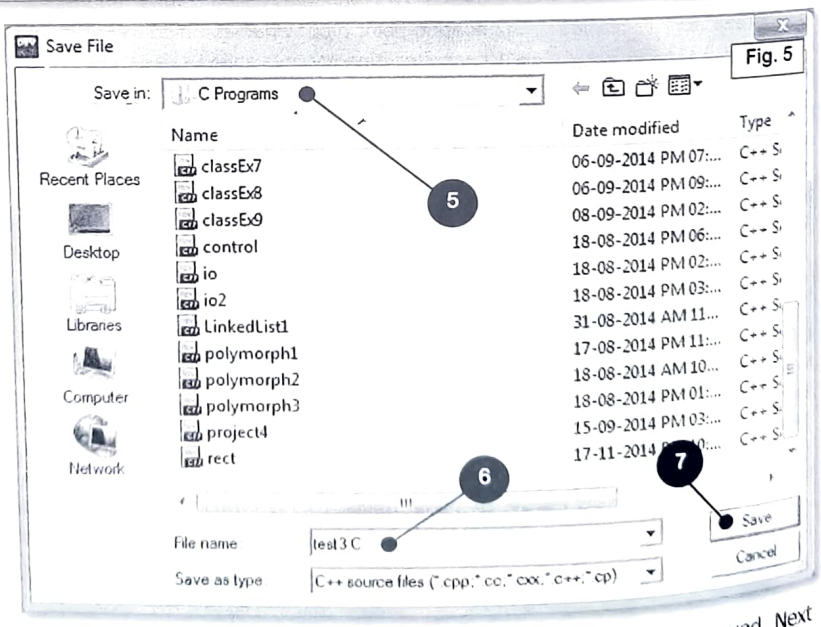
Click on **File** menu and from the dropdown list select **New→Source File** as shown in figure-2.

Type the code exactly as shown in figure-3 in the code area (you may not understand every bit of the code, but don't lose heart, everything will be clear to you by the time you complete the next chapter).



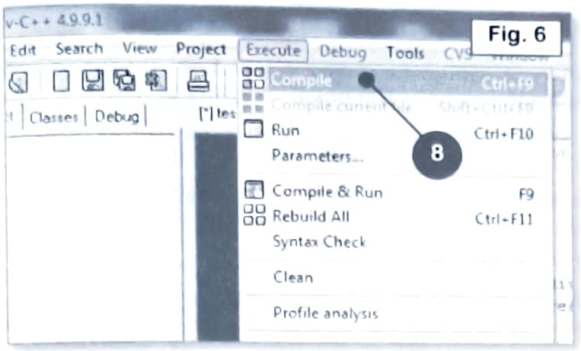


Once you have finished writing the code you must save the program. Click on **File** menu and from the dropdown list select **Save** as shown in figure-4.



The **Save As** dialogue box will open as shown in figure 5. From the **Save in** combo box select the folder where the file is to be saved. Next type the file name under the **File name** box. **Remember to put the extension as .C** like test3.C in the file name. Click on **Save** button to save the file.

Before you run the program it **needs to be compiled**.

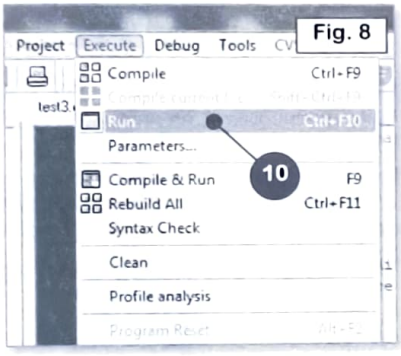




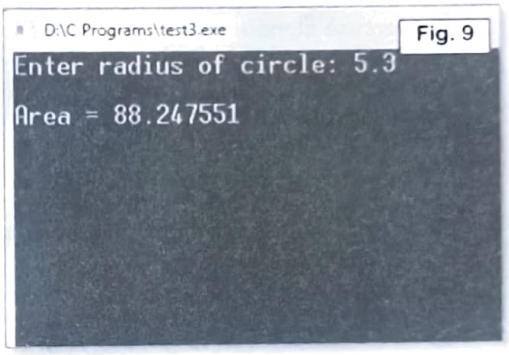
Click on the **Execute** menu and from the dropdown list select **Compile** as shown in figure-6. You can also press **Ctrl+F9** to compile a program.

The **Compile Progress** dialogue box will open as shown in figure-7. The compilation process will check the correctness of the code with respect to the syntax and include the contents of the header files, add the function libraries used in the program and convert the source code to a binary object code. If the file is successfully compiled the status will be shown as 'Done'. Click on the **Close** button to close the dialogue box.

To **Run** the program, click on the **Execute** menu and from the dropdown list select **Run** as shown in figure-8.



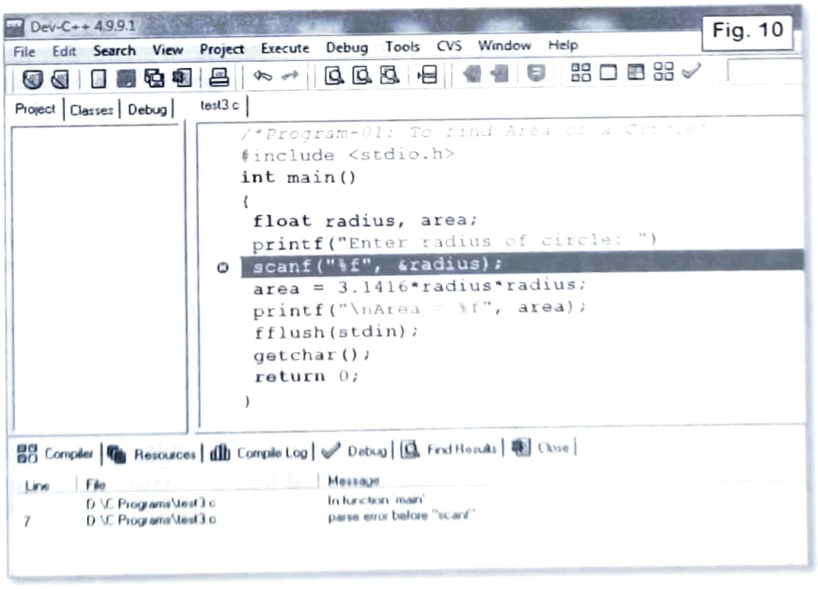
A black window will open as shown in figure-9. It will display the lines that you had written for output purpose and take inputs wherever required.



In the above program the code will first ask to enter the radius of the circle. A cursor will blink prompting the user to enter the radius as 5.3. The program next calculates the area and displays the result as 88.247551.

In case there is some error in the program, the compilation process will indicate the error.

Figure-10 shows such a situation. The semicolon is missing from line 6. The error is detected in line-7 and the line is highlighted. At the bottom of the window a message is displayed indicating there is an error in the program in a particular line number (line-7).



You need to rectify the error and **Save** and re-Compile the code before you can run it again.

Note that the line in which the error is indicated may not be the same line where the actual error is present. That can be sometimes misleading also. The reason for this is that the line where the error is shown is actually the line where the effect of the actual error is detected. Hence try to find the error either in the line marked or at a nearby line before the marked line.

Common reasons for error include missing semicolons, improper use of parenthesis, misspelt keywords, variables or library function names, and putting a calculation on the left side of the equal symbol.

The Fact File

- **C is a language that stands in-between a high level and a low level language and is often called a Middle Level Language as it was designed to have the positive points of both the types of languages**
- Every C program consists of one or more modules called functions, which are a series of instructions to the computer to perform a specific task. Of these functions, the most important is the main( ) function
- **Like the words in a particular language that are used to write the sentences, C has its own set of reserved words or keywords that are used to construct a C program**
- The original C language supported 32 basic keywords
- **Identifiers are names that are used to identify the different parts of a program. These names can be given to things like variables, constants, functions, arrays, structures etc. that are used as the building blocks of a program**
- An identifier name can consist of the alphabets (A to Z, a to z), digits (0 to 9), and the underscore symbol (_) only. It cannot start with a digit and cannot be a keyword also. No other characters are allowed including a blank space
- **A function is a group of statements that perform some meaningful work and normally returns a result back to the point from where it was used**
- C is often called a free form language as there are no specific rules to position a statement. However proper indentations in writing a program are encouraged as these make the program structure easy to read and debug
- **Lowercase alphabets are used to denote any keyword, function names and variables, whereas capital letters are used to indicate constants**
- For a better understanding of the program logic, sometimes it may be helpful to add comments in a program. Accordingly the program name, programmer name, the purpose of the program, the program version and any other relevant information may be included in a program
- **The header file that we will use in almost all C programs is the stdio.h file (the initials stdio stand for "standard input output"). This header file contains all the information that the compiler needs for functions dealing with input and output operations and working with the disk, monitor and printer**
- An opening brace " { " must appear before the first instruction after the function name and a closing brace " } " must follow the last instruction of the function. The opening and closing braces are called delimiters and they mark the beginning and end of a block of code
- **Each instruction in a C program must end with a semicolon ";" which is also known as a statement terminator. The semicolon tells the compiler that it has reached the end of the instruction and what comes next will be another instruction or the end of the program**
- An integer number is a number with no decimal point i.e. a whole number. It can be a positive or negative number or zero. It is used generally for counting purposes or for calculations involving only integer values
- **To represent real numbers i.e. numbers having both an integer part and a fractional part, a computer typically stores the real number as a combination of three parameters viz. the sign, mantissa, and an exponent part**
- A double data type is also used to represent real numbers i.e. numbers having both an integer part and a fractional part. It is also called a double precision floating point number. This is used for variables that require greater precision
- **A character type data represents a single letter, numeral or other keyboard characters. Character type data can be used where the input required is in the form of a single character, like selection from a set of multiple choices**
- In case a data is of a basic data type i.e. int, float, double or char it is called a primary data type
- **Apart from the basic data types, data can also be of type array, pointer, structure, union or enum. Such a data type is called a secondary constant or variable**
- You can modify the range of the primary data types using a set of keywords called data modifiers. They define the different amounts of storage that are possible for a variable of a given data type
- **Each type of data can be broadly divided into a constant or a variable depending upon the way it is being used. As the name suggests the value of a constant remains the same or constant throughout the program. Whereas the value stored in a variable can change or vary in the course of a program**
- The #define directive creates symbolic constants i.e. constants represented as symbols and macros. A symbolic constant has to be defined before the main( ) function
- **A literal is any piece of data that one directly types in into a C program. It can be any number, character or string that one may use as an initial value in the program**
- Variables may have an initial value i.e. one may want a variable to have a certain value at the start of a program, though this value can change as the program runs

Review Questions

Q1. **Multiple Choice Questions. Select from any one of the four options.** **1 each**

i) The most important function in a C program is called:
a. imp b. entry c. start d. main

ii) To write a program, C has its own set of reserved words also called:
a. identifiers b. literals c. main words d. key words

iii) Names used to identify the different parts of a C program are in general called:
a. identifiers b. specifiers c. functions d. literals

iv) A identifier in C cannot contain which of the following characters?
a. alphabet b. digit c. blank space d. underscore

v) A piece of information that does not run but used for a better understanding of a program:
a. literal b. function c. comment d. keyword

vi) The files that need to be included in a C program to use various library functions, are called:
a. header files b. important files c. basic files d. library files

vii) The initials stdio for the header file stdio.h stand for:
a. shell input output b. storage input output c. standard input output d. studio input output

viii) Each instruction in a C program must end with a:
a. full stop b. comma c. semicolon d. colon

ix) Which of the following is not a primary data type in C?
a. float b. double c. char d. enum

x) Which of the following data types is not used to represent real numbers?
a. long double b. float c. double d. char

xi) Which of the following represents a whole number type data in C?
a. long b. float c. double d. long double

xii) What is the precision of a float type data in C?
a. 7 decimal places b. 10 decimal places c. 15 decimal places d. 19 decimal places

xiii) What is the precision of a double type data in C?
a. 7 decimal places b. 10 decimal places c. 15 decimal places d. 19 decimal places

xiv) What is the precision of a long double type data in C?
a. 10 decimal places b. 15 decimal places c. 19 decimal places d. 22 decimal places

xv) What is the minimum value that can be stored in a 2 byte signed integer type data in C?
a. 0 b. -32767 c. -255 d. -32768

xvi) A signed char type data has a range of _____:
a. 0 to 255 b. -128 to +127 c. 0 to 128 d. 0 to 256

xvii) An unsigned integer which is 2 bytes long can store values in the range _____:
a. 0 to +32767 b. 0 to +65535 c. 0 to +255 d. 0 to +65536

xviii) A temporary storage location for data that can change in a program is called a _____:
a. constant b. literal c. variable d. macro

xix) Which of the following is not a valid octal type data in C?
a. 00 b. 1234 c. 028 d. 067

xx) Which of the following is not a valid hexadecimal type data in C?
a. 0xAB b. 0xx c. 0xA d. 0XB

xxi) #define is an example of a _____ directive:
a. variable b. declaration c. functional d. pre processor

Q2. **Short Answer type questions:** **1 each**

i) Name any one header file used in C.

ii) What is a keyword?

iii) What do you mean by an identifier in C?

iv) What do you mean by a constant in C?

v) What do you mean by a variable in C?

vi) State any one method of defining a constant type data in C.

vii) Name any two primary data types in C.

viii) State the range of a 2 byte integer type data.

ix) What is the purpose of a data modifier in C?

x) Name any two data modifiers in C.

xi) What is the range of a signed char type data?

xii) What is the difference between a floating point type data and a double type data?

xiii) What is the use of the 'const' keyword in C?

xiv) State the precision of a long double type data in C.

xv) What do you mean by a literal in C?

Q3. **Long Answer type questions:** **7 each**

i) What is the difference between the statements #include <file.h> and #include "file.h" in C? Discuss any two building blocks or tokens in a C program. Give an example of a hexadecimal constant. 2+2+2+1

ii) What is a variable? Explain the two different ways of defining a constant value in C. What is the range of an unsigned char type data in C? 2+2+2+1

iii) State the rules of naming an identifier in C. Discuss on an int and a float type data in C. 3+2+2

iv) What do you mean by declaring and initialising a variable in C? Give proper examples. Explain with proper examples the purpose of using data modifiers in C. Give an example of an octal and a hexadecimal constant in C. 2+2+2+1

v) What do you mean by a literal in C? What is the purpose of including the stdio.h file in a C program? Write the C code to define two constants called **factor** that refers to the value **2.54** and another one called **wish** that refers to the value "**Good Morning**". 2+2+2+1

CHAPTER 9
Input / Output and Simple Calculations in C

- Introduction 9-1
- Formatted text output: the printf( ) function 9-1
- Formatted text input: the scanf() function 9-3
- More on the use of Format Specifiers and Escape Sequences 9-6
- Unformatted text output: the putchar( ) and puts( ) functions 9-11
- Unformatted text input: the getchar( ) and gets( ) functions 9-12
- Some Basic Operators and Simple Calculations 9-13
- Operators and Data Types 9-15
- Some worked out examples 9-19

9.1 Introduction

This chapter deals with the different functions that are used in C to carry out **basic input and output operations**. As has been discussed earlier, unlike some other programming languages, C does not have inbuilt keywords (like INPUT, PRINT etc.) that can be used to output data onto the screen, printer, or hard-disk and input data from the keyboard, or hard-disk.

Input and output can be from the **standard input/output devices** like the keyboard and monitor or can be from files. **Input/output from files** is dealt with in class 12. This chapter deals with the standard input and output functions that you have already used in the previous chapter.

Input and output can be both formatted and unformatted.

Input and output can be **both formatted and unformatted**. Formatting means, the input/output data is displayed on the screen depending upon the data type. For example integer data and floating point data are formatted differently. A floating point data can even be displayed with the number of digits after the decimal point as desired by the user. On the other hand unformatted data like a string can be displayed without applying any extra formatting. All these things are discussed in detail in the following sections.

9.2 Formatted text output: the printf( ) function

The printf( ) function

An output operation sends a copy of the data stored in the computer's memory to another location like a monitor, printer, disk etc. without erasing the data from the computer's memory or changing the way the data is stored. The most important of these output functions in C is the versatile `printf()`, which can **display all types of data** and can work with multiple arguments. In addition, `printf()` can **format the way the data appears**.

printf( ) function can display all types of data and can work with multiple arguments and format the data.

The data that needs to be displayed on the screen forms the argument of the `printf()` function and is to be placed within the pair of braces `()` after the `printf` function name. In order to format and display all types of data, the `printf()` function argument is divided into two parts.

- The **first part** is called the **control string** or format string
- The **second part** is called the **data list**

The **control string** indicates how and where the data is to be displayed. It includes **within double quotes** any text that is to be displayed along with special characters called **format specifiers**. The format specifiers function as **placeholders** and leave space normally for displaying any value from a constant or a variable. Each format specifier starts with a `%` sign followed by a letter indicating the data type, as shown below:

Types of Format Specifiers

Specifier	Data Type	Range	Bytes	Example
%d	Signed integer	-32768 to +32767	2	25
%ld	Long signed integer	-2147483648 to +2147483647	4	-2547896
%u	Unsigned integer (only positive)	0 to 65535	2	55525
%lu	Unsigned long integer (only positive)	0 to 4294967295	4	958478625
%f	Float	-3.4e38 to +3.4e38	4	12.051000
%lf	Double	-1.7e308 to +1.7e308	8	5.368e
%Lf	Long double	-1.7e4932 to +1.7e4932	10	-1.5e400

Specifier	Data Type	Range	Bytes	Example
%e	Scientific notation	-3.4e38 to +3.4e38	4	1.205100e+01
%g	Shortest decimal /scientific notation	-3.4e38 to +3.4e38	4	12.051
%c	Character	0 to 255	1	'C'
%s	String	N.A.	N.A.	"New String"

The **second part** of the argument i.e. the **data list** can contain one or more data items that are displayed using the control string. The data list is **separated** from the control string by a comma. When the program is run, the **placeholders in the control string are replaced by the data items in the date list**. The general format of a `printf()` function is:

Syntax of printf( ) function

```
printf("control string with placeholders", data1, data2, ... );
```

The following example shows how the `printf()` function works by displaying the value stored in the variable `age` in the position indicated by the placeholder `%d` in the control string.

```
int age = 17;
printf("I am %d years old", age);
```

When run, the output of the above statement will be:

```
I am 17 years old
```

Characteristics of printf( )

Some characteristics of the `printf()` function are given below:

- There should be a **single control string** in a given `printf()` function call
- A `printf()` function may **only have a control string**, without any data list
- There can be none, or **multiple data items** in the data list part of `printf()`
- In case there are multiple data items in the data list, a **format specifier** must be included in the control string **for each data item in the list**
- The **values in the data list must appear in the same order as their specifiers** in the control string. Thus the first item in the list is substituted for the first format specifier, the second item for the second format specifier and so on

The examples given below illustrate the various uses of the printf() function.

Program Line	Output Displayed
`printf("The value of PI is %f", 3.1416);`	`The value of PI is 3.1416`
`int a = 3;` `printf("The square of %d is %d", a, a*a);`	`The square of 3 is 9`

`int x=7;`
`float y=5.5, p;`
`p = x*y;`
`printf("Product of %d and %f is %f", x, y, p);`
	`Product of 7 and 5.500000 is 38.500000`

`char c1='J', c2='B';` `printf("My initials are %c & %c", c1, c2);`	`My initials are J & B`
`int num=65;` `printf("The ASCII of %c is %d", num, num);`	`The ASCII of A is 65`
`printf("%%d is used to format an int");`	`%d is used to format an int`

In case of a mismatch between specifier type and data type, proper output will not be displayed

In all the above examples, care has been taken such that **the order in which the format specifiers appear, match with the data and data types in the data list**. Thus in the example:

Multiple outputs using printf()

```
printf("Product of %d and %f is %f", x, y, p);
```

`x` has to be an integer type variable and `y` and `p` have to be float type variables to match the declarations in the control string, so that the first `%d` will display the value in `x`, the first `%f` will display the value in `y` and the second `%f` will display the product value in `p`.

9.3 Formatted text input: the scanf( ) function:

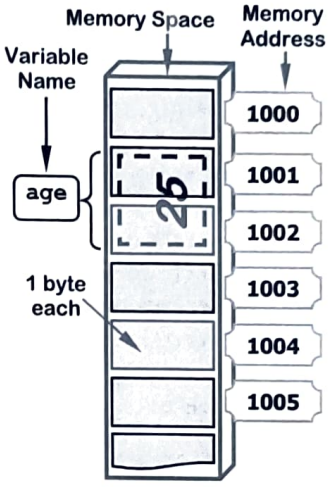
The scanf( ) function

Input is the process by which **data is fed into the computer for use by the program**. This data input to the computer is **stored in a variable** to be used when required by the program. The data input into a variable can change each time a new input is given to the same variable. We will discuss here the most common input function that is used for inputting formatted data i.e. the `scanf()` function.

`scanf()` is the general purpose input function which **can be used to input all types of data** from the keyboard. `scanf()` **scans** formatted data from the keyboard. The function scans the keyboard for any keystroke and then interprets the data input based on the format specifiers used. One `scanf()` function can take different types of data all at the same time.

To understand how `scanf()` works let us first understand how a variable is stored in the computer. For example consider the variable declaration.

```
int age = 25;
```



When such a variable is declared then some memory space is reserved for storing the value when the program is run. If we consider the basic unit of memory to be a byte, then for storing the above variable i.e. `age`, minimum 2 bytes of memory are reserved. To identify the location where this memory space is reserved, each byte is assigned a memory address as shown in the diagram on the right. In our example, the memory locations 1000 and 1001 (2 bytes) are reserved for storing the variable and the **variable name `age` is associated with these memory locations**. Hence whenever we refer to the variable name `age`, we indirectly refer to the above memory locations.

The address of the variable `age` is taken as 1000 (though two bytes are reserved but only the first byte is taken as the address of a variable always). If the variable `age` is initialised with the value 25, as shown above, then the memory locations 1000 and 1001 will hold the value 25 as seen from the diagram.

The value can also be entered by the user from the keyboard using the `scanf()` function during running the program, as shown in the following example:



Components of scanf( )

The data that needs to be input forms the argument of the `scanf()` function. In order to input data, just like the printf() function the `scanf()` function argument is divided into two parts.

- The **first part** is the **control string** or format string
- The **second part** is the **data list**

The **control string** indicates how the input data is to be formatted. Similar to the `printf()` function, the **control string** contains the format specifiers, which are known as **conversion characters** as they determine how the data is to be interpreted during input. In the above example, the %d format specifier within the control string indicates that the data input is to be interpreted as a signed integer.

The address operator '&' is to be used with the majority of data types in C.

The **second part** of the argument i.e. the **data list** contains the **address references** of one or more data items that are input by the user. The data list is **separated** from the control string by a comma. The general format of a `scanf()` function is:

```
scanf("control string with placeholders", data1, data2, ... );
```

Control string of scanf( )

For the majority of data types (except arrays and pointers, to be discussed later) the address reference of a variable is denoted by placing the **address operator** '`&`' before the variable name. The address operator refers to the memory address at which the contents of the variable are stored. Therefore `&age` refers to the memory address at which the content of the variable `age` is to be stored (1000 in this example). **Remember that the data list in `scanf()` contains the addresses of the variables** (and **NOT** just the variable names) at which the input data is to be stored.

When using **multiple inputs** through a single `scanf()` function one important thing should be kept in mind. In the `scanf()` function the **number and type of conversion characters in the control string should exactly match or correspond to the number and type of variable names in the data list**. The following example will make the statement clear:

```
int age;
float weight, height;
scanf( "%d %f %f" , &age, &weight, &height );
```

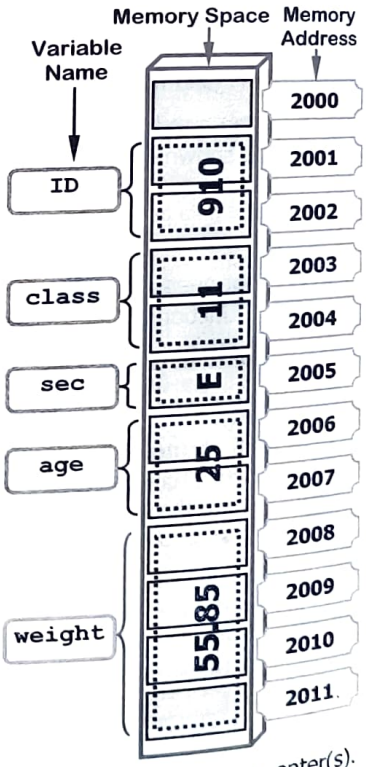
scanf( ) with multiple inputs

The following program displays the use of the `scanf()` function with multiple inputs.

```
1  /*Program-02: Use of scanf() function*/
2  #include <stdio.h>
3  int main()
4  {int ID, class, age;
5   char sec;
6   float weight;
7   printf("Enter your ID, Class, Section, Age, and Weight in order:");
8   printf("\nSeparate each piece of data with a blank...");
9   scanf("%d%d%c%d%f", &ID, &class, &sec, &age, &weight);
10  printf("\nYour ID=%d, Class=%d and Section=%c", ID, class, sec);
11  printf("\nYour Age=%d years and weight=%f Kg", age, weight);
12  return 0;
13 }
```

Output:

```
Enter your ID, Class, Section, Age, and Weight in order:
Separate each piece of data with a blank...
910  11  E  17  55.85
Your ID=910, Class=11 and Section=E
Your Age=17 years and weight=55.85 Kg
```



The control string in scanf( ) is used to format the input data as an int, float, char, double, etc.

Note the following regarding this program and the use of `scanf()`.

- Firstly, in the control string the conversion characters should be written without any blank or any character in-between them to avoid the chance for any compilation error. Thus `"%d%d%c%d%f"` is desirable than `"%d %d %c %d %f"` (however extra characters can be put inside the control string as will be shown later).
- Secondly, we are formatting the variables ID and class as integers. The first `%d` corresponds to `ID`, the second `%d` corresponds to `class`. Next the section variable `sec` is input through the `%c` conversion character. Finally `age` and `weight` are input as `int` and `float` respectively.
- Thirdly, the address operator '`&`' is used along with the variable names for all the inputs i.e. `ID` (`int`), `class` (`int`), `sec` (`char`), `age` (`int`), `weight` (`float`). We will see in the chapter dealing with strings, how to input a string using the `%s` specifier (the '`&`' operator is not used with a string variable).

When multiple inputs are done using a single scanf() function, during input each piece of data should be separated by a white space character

- **When more than one input** is there in the `scanf()` function, during input each piece of data should be separated by a blank(s) or a tab(s) or by an enter(s). `scanf()` **ignores any such white-space character** or characters such as blank, tab or enter and considers as valid inputs only non-white-space characters. Thus **each piece of data can be separated by more than one blank, tab or enter, with no effect** on the input as long as the data types match with the conversion character types.

- In the above diagram note how memory is allocated to store the variables. All integer variables are allotted 2 bytes of memory, floating point variable 4 bytes of memory and character type variable 1 byte of memory. In this example the variable `ID` has the memory location 2001, `weight` has the memory location 2008 etc.

Be careful in specifying the order in which the conversion characters and the variable names are written. If the order does not match, then proper data will not be input into the variables.

Note how the **newline escape sequence \n** is used in lines 8, 10, and 11 to print text from a new line.

One more thing needs to be noted. During declaration a `char` type data can be declared as an `int` but during input the `%c` **specifier should be used in** `scanf()` if the data is to be used as `char` type later.

• Using other characters in the control string of scanf( ):


Using other characters in control string

Though it is not preferable to use characters other than the conversion characters in the control string of the `scanf()` function but **extra characters can be used** in the control string in the way as shown in the following C program.

The following program displays the use of the `scanf()` function with additional control string characters.

```
1  /*Program-03: Use of extra characters within scanf() control string*/
2  #include <stdio.h>
3  int main()
4  {float rate, amount;
5   int qty;
6   printf("Enter Rate of Item per Kg in Rs.: ");
7   scanf("Rs.%f", &rate);
8   printf("Enter quantity of Item in Kg: ");
9   scanf("%dKg", &qty);
10  amount = rate*qty;
11  printf("\nThe total amount = Rs. %f", amount);
12  return 0;
13 }
```

Output:

```
Enter Rate of Item per Kg in Rs.:
Rs.56.50                          ⇒ Value is entered as Rs.56.50 and not as 56.50 only
Enter quantity of Item in Kg:
10Kg                              ⇒ Value is entered as 10Kg and not as 10
The total amount = Rs. 565.000000
```

In the above example the literal string "`Rs.`" in the control string for `scanf()` in line-7 indicates that during input the user should type the rate along with the currency sign i.e. `Rs.56.50` for example. Similarly in line-9 the literal string "`Kg`" in the control string indicates that the input data value should be followed by the characters "`Kg`". This means that the `scanf()` function while interpreting the input stream will expect the "`Rs.`" string before the `float` input in line-7 and the "`Kg`" string after the integer input in line-9. Accordingly it will ignore these and convert whatever value is input *after* the string literal in line-7 as per the `%f` conversion character and whatever value is before the string literal in line-9 as per the `%d` conversion character as valid input.

While using such string literals care should be taken so that **the user types in exactly what has been put as the string literal**, otherwise unpredictable input will take place. Thus in the above example though the user makes the input as '`Rs.56.50`' but the value contained in the variable `rate` will be only '`56.50`'.

• A peek into the working of the scanf( ) function and use of fflush(stdin):


Working of scanf( )

Let us now examine how the `scanf()` function reads the data during input. Data is read by the `scanf()` function from what is called the **input stream**, which means a series of characters being input from some source. In the case of `scanf()` it is the raw data that is input from the keyboard. After typing the input when the *Enter* key is pressed, the data that you have typed is sent to the `scanf()` function as a series of

Conversion characters

meaningless characters in the order in which it was typed. The `scanf()` function then reads the data and uses the format specifiers or **conversion characters** to **convert** the raw data to the desired data types which agree with the conversion characters.

During conversion it ignores all white-space characters and looks for the first non-white-space character which matches with the specifier, and assigns it to the respective variable. The assignment stops at the first non-matching character.

Always use the fflush(stdin) statement before entering a character type or string type data.

Important Note!: The `scanf()` function uses an area of the memory called the **input buffer** to store the characters being typed into the input stream, before converting the data using the conversion characters. As an example when an integer is input from the keyboard, the user types the integer and presses the Enter key. The integer along with the white-space character 'Enter' is then passed to the input buffer. The `scanf()` function then interprets and converts the data input as per the conversion character within its control string.

In case of the integer data, it will take the characters up to the next white-space character and leave the white-space character 'Enter' in the input buffer. In case the next data to be input is a character or string type data, then `scanf()` will first read the white-space character still remaining in the input buffer as a leftover from the previous input. Unlike a numeric data, this white-space character however is a valid input for a character or string type data. Thus instead of using the current data as an input, it will take the leftover white-space character from the last input as a valid input. This causes an error in the input and the program will not wait for the current input from the user as it takes the leftover white-space from the input buffer.

The fflush( ) function

To avoid this problem the `fflush()` function is used which **flushes out or clears** any data that may remain in the buffer. The argument of the `fflush()` function is the **buffer name** that we want to clear. In this case the argument should be the buffer related to the **standard input device** i.e. the **keyboard** and which is designated by '`stdin`' (**st**andard **in**put device). For example to input a `char` type data `x`, the code will be:

```
fflush(stdin);
scanf("%c", &x);
```

Always use the `fflush(stdin)` statement before entering a character type or string type data using the `scanf()`, `getchar()` or `gets()` functions to avoid any malfunction

9.4 More on the use of Format Specifiers and Escape Sequences

This section deals with the various aspects of using the format specifiers in a program. The following program shows the use of the `printf()` and `scanf()` functions along with the different format specifiers and conversion characters as discussed in page P1-9-1.

Use of different format specifiers

% symbol can used before a mat specifier ignore it as a mat specifier de the control ng

```
1   /*Program-04 to display the use of different format specifiers*/
2   #include<stdio.h>
3   int main()
4   {int num1; float num2; char ch;
5    printf("\nEnter an integer (Number1<=32767): ");
6    scanf("%d", &num1);
7    printf("\nEnter a floating point number (Number2): ");
8    scanf("%f", &num2);
9    printf("\nEnter a character: ");
10   fflush(stdin);
11   scanf("%c", &ch);
12   printf("\nNumber1 printed with a %%d specifier: %d", num1);
13   printf("\nNumber1 printed with a %%u specifier: %u", num1);
14   printf("\n\nNumber2 printed with a %%f specifier: %f", num2);
15   printf("\nNumber2 printed with a %%e specifier: %e", num2);
16   printf("\nNumber2 printed with a %%g specifier: %g", num2);
17   printf("\n\nCharacter entered printed with a %%c specifier: %c", ch);
18   return 0;
19  }
```

Output Set-1

```
Enter an integer (Number1<=32767): 23456
Enter a floating point number (Number2): 6452.76
Enter a character: S

Number1 printed with a %d specifier: 23456
Number1 printed with a %u specifier: 23456

Number2 printed with a %f specifier: 6452.759766
Number2 printed with a %e specifier: 6.452760e+003
Number2 printed with a %g specifier: 6452.76

Character entered printed with a %c specifier: S
```

⇒ Integer number within range
⇒ Floating point number within range
⇒ Character data input (single character)
⇒ Integer displayed correctly using %d
⇒ Integer displayed correctly using %u
⇒ Displayed as a floating point number
⇒ Floating point in scientific notation
⇒ Expressed in shortest decimal notation
⇒ Displayed as a character



Use of format specifiers

Output Set-2

```
Enter an integer (Number1<=32767): -2345
Enter a floating point number (Number2): 0.009876
Enter a character: Jashojit

Number1 printed with a %d specifier: -2345
Number1 printed with a %u specifier: 63191

Number2 printed with a %f specifier: 0.009876
Number2 printed with a %e specifier: 9.876000e-003
Number2 printed with a %g specifier: 0.009876

Character entered printed with a %c specifier: J
```

⇒ Integer number within range
⇒ Floating point number within range
⇒ Multiple-character input
⇒ Integer displayed correctly using %d
⇒ Being signed, displayed incorrectly using the %u (unsigned) specifier
⇒ Displayed as a floating point number
⇒ Floating point in scientific notation
⇒ Expressed in shortest decimal notation
⇒ Only the first character is displayed though multiple characters were input

Output Set-3

```
Enter an integer (Number1<=32767): 45678
Enter a floating point number (Number2): 0.0
Enter a character: 5

Number1 printed with a %d specifier: -19858
Number1 printed with a %u specifier: 45678

Number2 printed with a %f specifier: 0.000000
Number2 printed with a %e specifier: 0.000000e+000
Number2 printed with a %g specifier: 0
Character entered printed with a %c specifier: 5
```

⇒ Integer number out of range of integers
⇒ Floating point number within range
⇒ Single digit input as a character
⇒ Out of range integer displayed incorrectly
⇒ Within range of unsigned integers hence displayed correctly using %u
⇒ Displayed as a floating point number
⇒ Floating point in scientific notation
⇒ Expressed in shortest decimal notation
⇒ Digit displayed as a character

Observe the use of the function `fflush(stdin)` in line-10 of the last program, before the character input. The next program shows another use of the `%c` format specifier and `fflush()` function.



Printing ASCII value of a character

```
1  /*Program-05 To display the ASCII value of a character*/
2  #include<stdio.h>
3  int main()
4  {char ch;
5   printf("Enter the character to find its ASCII value: ");
6   fflush(stdin);
7   scanf("%c", &ch);
8   printf("\nThe ASCII value of %c is %d", ch, ch);
9   return 0;
10 }
```

Output of the above program:

```
Enter the character to find its ASCII value: C
The ASCII value of C is 67
```

Two different specifiers are used to display the same character variable `ch`. **Each character input from the keyboard has an 8 bit integer value associated with it**. The computer stores this integer value to represent the character. This value is known as the American Standard Code for Information Interchange or **ASCII code of the character**. Here the `%d` specifier, which displays an integer, converts the character represented by `ch` to the corresponding ASCII value and displays the value. This statement can be used to display all the ASCII characters and their corresponding ASCII values using proper looping (we will write such a program in a later chapter). **Thus we can change the displayed output by changing the format specifier for the same variable.**



Use of the %s specifier

The following program displays the use of the `%s` format specifier.

```
1  /*Program-06 to display the use of the %s specifier*/
2  #include<stdio.h>
3  #define WISH "Thank You"
4  int main()
5  {char Name[ ] = "Anuraj";
6   printf("Using %%s specifier to print: %s\n", WISH);
7   printf("Using %%s specifier to print: %s\n", Name);
8   return 0;
9  }
```

Output of the above program:

```
Using %s specifier to print: Thank You
Using %s specifier to print: Anuraj
```

A detailed discussion of the use of the `%s` format specifier **used to handle strings** will be done in the chapter dealing with strings. So do not get disappointed by the syntax of the above program. For the time being just keep in mind how the `%s` specifier is used in the above example. We have used `%s` in line-6 to **display a constant string** defined in line-3 using the `#define` statement. Line-7 **displays a variable string** that is initialised in the character array (this is how a string is represented in C) in line-5.

Two more format specifiers of interest are the `%o` (not zero, but 'o') and the `%x` specifiers which convert an integer type data to the corresponding **Octal** and **Hexadecimal** number respectively. The following example shows the use of these specifiers.



Displaying Octal and Hexadecimal values

The following program displays the use of the `%o` and `%x` format specifier.

```
1  /*Program-07 to display the use of the %o and %x specifiers*/
2  #include <stdio.h>
3  int main()
4  {int num;
5   printf("\nEnter an integer number: ");
6   scanf("%d", &num);
7   printf("\nThe number %d is %o in Octal and %x in Hex", num, num, num);
8   return 0;
9  }
```

The spacing and the number of digits/characters displayed are controlled by the field width specifiers

Output of the above program:

```
Enter an integer number: 14
The number 14 is 16 in Octal and E in Hex
```

Note that the value stored in the same variable `num` is displayed in three different manners using three different format specifiers in line-7.



Field width specifiers

- Field Width Specifiers:

As stated earlier, the format specifiers are mainly used **for formatting the way the data is displayed**. This is done using the **Field Width Specifiers**. The **spacing** and the **number of digits/characters displayed** are controlled by the field width specifiers. The general format for a field width specifier is:



N reserves the **number of spaces allotted for a field**. In case the data length is less than N, **extra spaces will be put** to make the total as **N**. This does not imply that the full data will not be displayed if the actual data length is more than the specified length. For float type data it **includes the decimal point** in the width. In such case **C ignores the field width specified and displays the whole data**.

However the **meaning of n** depends upon the **data type**. For `float` type data it indicates the number of places to display after the decimal point. For `int` type data it represents the number of digits to display with leading zeroes if the value of **n** is more than the number of digits. For string type data it indicates the number of characters to display.

Certain **special symbols** when inserted along with the field width specifiers can align (justify) the data along certain lines or put extra characters along with the data. These are:

- **0** (zero): This causes **leading zeroes** to come **in place of blank spaces** before the data, if the specified field width is longer than the actual data width.
- **–** (minus): This causes the data item to be **left justified** within the specified field. Thus **blank spaces** to fill the remaining field width will be **added after the data** and not before it.
- **+** (plus): This causes a **sign to precede each signed numerical data** item i.e. depending upon the value of the data, either a positive or a negative sign will precede the data.
- **#** (hash): When using an e, f or g type conversion this causes a **decimal point to be present in all floating point numbers** even if the data item is a whole number. It also **prevents the truncation of trailing zeroes** in g type conversion.

Field width specifiers

For real numbers, the decimal point is included with the width.

Special symbols with field width specifiers

The following examples along with the outputs clarify the use of field width specifiers with **float type data**:

Examples of field width specifiers

```
printf("Cost is Rs%f:", 26.39);          Output ⇒ Cost is Rs26.390000:
printf("Cost is Rs%.2f:", 26.39);        Output ⇒ Cost is Rs26.39:
printf("Cost is Rs%8.3f:", 26.39);       Output ⇒ Cost is Rs   26.390:
printf("Cost is Rs%-8.2f:", 26.39);      Output ⇒ Cost is Rs26.39   :
printf("Cost is Rs%5.0f:", 26.39);       Output ⇒ Cost is Rs   26:
printf("Cost is Rs%08.2f:", 26.39);      Output ⇒ Cost is Rs00026.39:
printf("Profit is Rs%+8.2f:", 26.39);    Output ⇒ Profit is Rs  +26.39:
printf("Profit is Rs%-+8.2f:", 26.39);   Output ⇒ Profit is Rs+26.39  :
printf("Net charge is :%#6.0f:", 26.39); Output ⇒ Net charge is :26.   :
printf("Cost is Rs%.*f:", 4, 26.39);     Output ⇒ Cost is Rs26.3900:
printf("Cost is Rs%*.*f:", 8, 4, 26.39); Output ⇒ Cost is Rs 26.3900:
```

Integers, floats, and strings can be formatted using Field Width Specifiers.

The following examples along with the outputs clarify the use of field width specifiers with **int type data**:

```
printf("Distance=%10d:meters", 215);     Output ⇒ Distance=       215:meters
printf("Distance=%-10d:meters", 215);    Output ⇒ Distance=215       :meters
Printf("Distance=%10.6d:meters", 215);   Output ⇒ Distance=    000215:meters
Printf("Distance=%-10.6d:meters", 215);  Output ⇒ Distance=000215    :meters
printf("Distance=%*d:meters", 10, 215);  Output ⇒ Distance=       215:meters
printf("Distance=%*.*d:meters",8,4,215); Output ⇒ Distance=    0215:meters
Printf("Net charge is :%-+4d:", 26);     Output ⇒ Net charge is :+26 :
```

The following examples along with the outputs clarify the use of field width specifiers with **string type data**:

```
Printf("Name=%s:", "Byomkesh");          Output ⇒ Name=Byomkesh:
Printf("Name=%5s:", "Byomkesh");         Output ⇒ Name=Byomkesh:
Printf("Name=%10s:", "Byomkesh");        Output ⇒ Name=   Byomkesh:
Printf("Name=%-10s:", "Byomkesh");       Output ⇒ Name=Byomkesh   :
Printf("Name=%.4s:", "Byomkesh");        Output ⇒ Name=Byom:
Printf("Name=%6.4s:", "Byomkesh");       Output ⇒ Name=   Byom:
Printf("Name=%-6.4s:", "Byomkesh");      Output ⇒ Name=Byom   :
```

Note that in case the format specifier is expressed in the form %*.*, then the width and the number of digits to display can be given as parameters to the `printf()` function as:

```
printf("Cost is Rs%*.*f:", 8, 4, 26.39);
```

Here, from the data list the first data item **8** corresponds to **N**, and the second data item **4** corresponds to **n**. The third data item 26.39 represents the data to be displayed. These can be taken as variables also.



Escape Sequences

- Escape Sequences:

Apart from using format specifiers for formatting text, `printf()` can use **special codes** called **Escape Sequences** to control the way the cursor moves on the screen or insert/display some special characters.

Each escape sequence begins with a **backslash** "\" which identifies the character that follows as an escape sequence. When the compiler encounters the backslash, it does not display the next character but performs the function indicated by it. Though each sequence comprises of two characters (backslash and the sequence code) however it is **treated as a single character by C and occupies 1 byte**.

Escape sequences:
\n newline
\t tab
\r carriage ret.
\b backspace
\f form feed
\\ backslash
\" double quote
\' single quote
\0 null character

The different types of escape sequences along with examples are given below:

- **The newline character**: the sequence `\n` performs a **newline** command i.e. it inserts a new line and prints whatver is written after it starting from a new line. There is no need to put any blank space between the escape sequence and the preceding or following text.

```
printf("A\nB\nC");
```

`\n` inserts a line after displaying `A` and `B` on the screen

Output1:

```
A
B
C
```

Escape sequences control the way the cursor moves on the screen and are treated as single characters.

- **The tab character**: the sequence **\t** moves the cursor to the next **preset tab** stop. The tab escape sequence is used **to allign data or text along a particular vertical line**.

```
printf("0\t1\t2\t3\t4");
```

`\t` inserts a tab (fixed space) after displaying each digit on the screen

Output2:

```
0    1    2    3    4
```

The following example shows a **use of the tab escape sequence to print a part of a calendar**:

```
printf("S\tM\tT\tW\Tt\tF\tS");
printf("\t\t1\t2\t3\t4\5");
printf("6\t7\t8\t9\t10\t11\12");
printf("13\t14\t15\t16\t17\t18\19");
```

Output3:

```
S    M    T    W    T    F    S
          1    2    3    4    5
6    7    8    9    10   11   12
13   14   15   16   17   18   19
```

- **The return character**: the sequence `\r` performs a **carriage return**, i.e. moving the cursor to the **start of the same line** without moving down to the next line. Hence if anything is written after executing the `\r` sequence, it will **overwrite the existing text**.

```
printf("Left\rRight");
```

Output4: Only the word `Right` will be displayed as it will overwrite the word `Left`

```
Right
```

- **The backspace character**: the sequence `\b` moves the cursor just **one position to the left**. It is **non-destructive** and **does not erase any character** as it moves to the left. If after a backspace command, a newline command is given, the cursor moves to the next line **without inserting** a blank line.
- **The formfeed character**: the `\f` sequence **ejects** the current page when output is sent to the printer.
- **Displaying certain special characters**: to distinguish between a syntactical sequence and any special punctuation mark like ', ", \ etc. in the control string, the character to be displayed is combined with the escape sequence \.

```
printf("Press \"ENTER\" to start");
```
Output ⇒
```
Press "ENTER" to start
```

```
printf("\\n inserts a new line");
```
Output ⇒
```
\n inserts a new line
```

9.5 Unformatted text output: the putchar( ) and puts( ) functions:

Unformatted Text Input

The `printf()` function is a general purpose function to display any string with or without format specifiers along with data, onto the screen. Apart from `printf()` two other function which can display characters on the screen are the `putchar()` function and the `puts()` function. However these functions are unable to format the text to the extent that `printf()` does. Hence these can be used at places where formatting the data is not required. Since the formatting overhead is not there in these functions, these functions are also **faster than the `printf()` function.**

• The putchar( ) function:

The putchar( ) function

The `putchar()` function **displays a single character value on the screen.** The parameter of the `putchar()` function must be either a `char` literal, `char` constant or a `char` variable. It is used instead of `printf()` at places, where only a single character needs to be displayed. When using literals, the **character literal should be put inside single quotes** as **'A'**. Examples of the `putchar()` function are given below:

```
putchar('H');
```
(using char literal) **Output ⇒**
```
H
```

```
#define TRUTH 'Y'
void main()
 {putchar(TRUTH);}
```
(using char constant) **Output ⇒**
```
Y
```

```
void main()
{char initial;
 initial = 'J';
 putchar(initial);}
```
(using char variable) **Output ⇒**
```
J
```

```
putchar('\n');
```
Output ⇒ a new line, treating the escape sequence `\n` as a **single control character** and not as two separate characters \ and `n`

• The puts( ) function:

The puts( ) function

Whereas the `putchar()` function displays a character on the screen, the `puts()` function is used to **display a text string** on the screen. Similar to `putchar()`, the `puts()` function displays unformatted string **starting from a new line**. Though formatting cannot be done, but the **escape sequences** like `\n` or `\t` **can be used** along with the string as shown below:

```
1 #define WISH "Good Morning Darjeeling"
2 void main()
3 {puts(WISH);
4  puts("Good\t\tMorning\t\tDarjeeling");
5  puts("Arise, awake, and stop not\nTill your goal is reached");
6 }
```

Output

```
Good Morning Darjeeling
Good        Morning     Darjeeling
Arise, awake, and stop not
Till your goal is reached
```

Note that line-3 prints the constant string literal WISH, line-4 prints a literal string with tab sequences and line-5 prints another string with a newline '`\n`' character in between. While printing, the newline splits the text to be displayed in two different lines.

9.6 Unformatted text input: the getchar( ) and gets( ) functions



Using getchar( ) function

- The getchar( ) function:

Similar to the `putchar()` function which is used to **output** an unformatted character data, there is the `getchar()` function **to input any unformatted character data**. One can input the character either as a `char` or an `int` type.

Unlike the `scanf()` function, the `getchar()` function **does not appear at the beginning of a line** but is usually assigned to a variable with the '`=`' sign. One more thing to note about the `getchar()` function is that **it does not take anything as its argument** i.e. nothing is to be put inside the braces following the `getchar` function name. When the program encounters the `getchar()` function, it stops and waits for some input from the keyboard. After typing in the desired character the same is to be followed by the **ENTER** key to assign the input to the variable.

Since `getchar()` expects a single character as input, whenever a character is typed into the keyboard, (including a blank or simply an Enter) the function reads that character and assigns it to the variable. The `getchar()` function also **echoes** the character typed, i.e. it displays the character typed onto the screen.

The getchar( ) function can be used to pause the program output before the last return statement.

The following program shows the use of the `getchar()` function.

```
1   /*Program-08: Use of getchar() function*/
2   #include<stdio.h>
3   int main()
4   {int ascii;
5    puts("Input any character to see its ASCII Code: ");
6    fflush(stdin);
7    ascii = getchar();
8    printf("The ASCII value of %c is %d.", ascii, ascii);
9    puts("\nPress ENTER to continue");
10   fflush(stdin);
11   getchar();
12   return 0;
13  }
```

Output:

```
Input any character to see its ASCII Code: A
The ASCII value of A is 65
Press ENTER to continue
```

⇒ The program waits at this line for an input

In line-4, the variable `ascii` has been defined as an integer. In line-7 the `getchar()` function is used to input a character and assign it to the variable `ascii`. The `printf()` function in line-8 prints the character using the `%c` specifier and prints the ASCII value of that character using the `%d` specifier.

In line-11 the `getchar()` function is used in a different manner i.e. the function is used without assigning it to any variable. When the program encounters the function, as usual it waits for some input from the keyboard. However when something is typed from the keyboard, then instead of assigning it to any variable as in line-7, the program control passes onto the next line i.e. line-12 and the data entered is lost forever. The `getchar()` **function is used in this manner when someone wants to halt the program** until the user wants to proceed further, as with the "`Press any key to continue`" type of comment. To view the output of a program without going to the output screen, this function can be used before the `return 0` statement. **Note** that `fflush(stdin)` function is used in line-6 and line-10 before the `getchar()` function.

Two more functions that are similar in operation as the `getchar()` function are the `getch()` and the `getche()` functions. However these functions may not be available with all compilers. These functions are ***available with the Turbo C compiler***.

The gets( ) function:

The gets( ) function

Apart from the **scanf()** function the **gets()** function is used to input any type of unformatted text/string into a variable. The **gets()** function can be used to input any type of string data. Unlike the **scanf()** function it can even input a string with blank spaces separating the words in the input stream. **More of gets() will be discussed in the chapter dealing with strings.** Presently we give a very brief description and use of **gets()**. The argument of the **gets()** function contains only the string variable name (without any address operator i.e. &) as shown in the example below:

The following program shows the use of the **gets()** function.

The gets( ) function can be used to input a string type data.

```
1  /*Program-09: Use of gets() function*/
2  #include <stdio.h>
3  int main()
4  {char name[21];
5   puts("Enter your name (<=20 characters)");
6   fflush(stdin);
7   gets(name);
8   printf("\nYour name is %s", name);
9   return 0;
10 }
```

Output:

```
Enter your name (<=20 characters)
A. R. Rahman
Your name is A. R. Rahman
```

⇒ The user input

A string in C is basically a **character array**. Line-4 defines the character array variable called **name** to store the string. **Don't worry if you can't understand the use of gets() at this point**. All these will be discussed in detail in the chapter dealing with strings.

The **gets()** function waits for the user to input the data string. As the data is typed in character by character, the same gets echoed (i.e. displayed) on the screen. After writing the string, only when the **ENTER** key is pressed the typed characters get assigned to the variable **name**. As long as the **ENTER** key is not pressed if any character(s) is mistyped, one can use the backspace key to delete the unwanted characters. Only when the enter key is pressed will the string input be assigned to the variable. Finally in line-8 the entered string is displayed using the **printf()** function and the **%s** format specifier.

9.7 Some Basic Operators and Simple Calculations

Using basic operators

After an input is made, it needs to be processed to give any useful information as output. This **processing of data may involve mathematical operations. Operators are required to perform the calculations** that transform data into information. It is basically a symbol that tells the computer how to process the data. Based on their mode of operation, operators can be of different types viz. **arithmetic**, **assignment**, **relational**, **increment**, & **logical**. Let us discuss the **arithmetic** & **assignment operators** and the rest will be discussed later. The following operators perform the mathematical functions as indicated:

Arithmetic operators

Operator	Purpose	Example/Use	Remarks
=	**assignment**	`y = 5` **or** `y = a+b-c`	Used to assign a value to a variable
+	**addition**	`y = a + b`	For a=8, b=5, y=13
−	**subtraction**	`y = a - b`	For a=8, b=5, y=3
*	**multiplication**	`y = a * b`	For a=8, b=5, y=40
/	**division**	`y = a / b`	For a=8, b=5, y=1 (Performs integer division)
%	**remainder**	`y = a % b`	For a=8, b=5, y=3 (Gets the remainder)

When using an operator to perform mathematical calculations, a variable is usually used to store the result of the calculation. **The variable and the arithmetic operation are connected by an equal sign "=" also called the assignment operator**, with the variable on the left side and the calculation on the right side.

There can be any number of variables connected by the operators on the right side of the "=" sign, but **on the left there should be only the name of the variable**. E.g.

`area = 3.1415*r*r;` ⇒ Calculates the area of a circle by multyplying the constant `3.1415` with `r*r` (**Note: there is no exponential operator "^" in C**)

`interest=(p*r*t)/100;` ⇒ Finds the interest by multiplying `p`, `r` & `t` and then dividing by `100`



Assignment operator '='

The **assignment operator** is used to assign the result of a calculation or a value **from the right of the operator to the left.** All calculations should be done on the right hand side of the assignment operator. (This is **different from the equality operator**)

The '`%`' or **remainder operator** may be new to you. It is used to get the remainder of dividing two integer variables. The result is also an integer. For example if `a` and `b` are two integer variables and a=11 & b=4, then the value of `a/b` will be equal to 2 and `a%b` will be equal to 3 as the remainder of dividing 11 by 4 is 3.

When **more than one of the above operators** is present in a calculation, then C performs the calculations in a certain order. The following table gives the **precedence** and **associativity** of the arithmetic operators. **Operators with a higher precedence are evaluated before operators with a lower precedence** in an expression. When a group of operators of the same precedence are present in an expression, then **associativity indicates the direction in which the operators will be evaluated**. The following table with examples clarifies the above concept.



Precedence and Associativity of operators'

Precedence	Operators	Associativity	Example	

High ↑ Low	( )		`y = 24 / 3 * 2 % 7 * 3 / 3`
			`= (24 / 3) * 2 % 7 * 3 / 3`
			`= (8 * 2) % 7 * 3 / 3`
			`= (16 % 7) * 3 / 3`
			`= (2 * 3) / 3`
			`= (6 / 3) = 2`
				`y = 48-128/32*3-7*6/2+9%2`
				`= 48-128/32*3-7*6/2+9%2`
				`= 48-4*3-42/2+1`
				`= 48-12-21+1`
				`= 36-21+1`
				`= 15+1`
				`= 16`
				⇓
				Expression with operators with **mixed precedence**. Note that higher precedence operators are calculated first

	*, /, %	**Left to Right** (i.e. calculation is done starting from the left)		

	+, −	**Left to Right**	`y = 6 - 15 + 14`
			`= (6 - 15) + 14`
			`= (- 9 + 14) = 5`	

	=	**Right to Left**	Assignment operator has the lowest priority i.e. performed at the last	

The above rule depicts the hierarchy of operations with more than one operator. However this hierarchy can be overridden by using brackets, which has the highest priority. The following example clarifies the point.

```
y = (512-128)/(32*2)-4*(33/(2+9)%2)
  = (512-128)/(32*2)-4*(33/(2+9)%2)
  = 384/64-4*(33/11%2)
  = 6-4*(3%2)
  = 6-4*1
  = 6-4
  = 2
```

Note that, irrespective of the precedence the expressions within the brackets are evaluated first, based on their associativity.

The next program tries to find how much is to be added to make an integer (input by the user) exactly divisible by 5 and then find the new integer so formed.

Algorithm:

- Step1: Input the number
- Step2: Divide the number by five
- Step3: Find the remainder
- Step4: Subtract the remainder from 5 to get the required number to be added
- Step5: Add this result to the input number to get the new integer

```
1  /*Program-10: To find number divisible by 5*/
2  #include<stdio.h>
3  #include<conio.h>
4  int main()
5   { int num, newnum;
6     clrscr();
7     printf("\nInput number to check divisibility by 5: ");
8     scanf("%d", &num);
9     newnum = num + 5 - num % 5;
10    printf("\nThe next highest number to %d, divisible by 5 is %d", num, newnum);
11    printf("\nPress any key to exit\n");
12    getch();
13    return 0;
14  }
```

In the above example in line-9, the calculation for **newnum** is done in the following order for an input value of **num** as **18**:

```
newnum = num + 5 - num % 5     ⇒ As per precedence first modulo or remainder calculation is done
       = 18 + 5 - 18 % 5
       = 18 + 5 - 3            ⇒ Second addition is done following the associativity of left-to-right
       = 23 - 3                ⇒ Finally subtraction is done
       = 20                    ⇒ To get the correct number 20
```

If precedence of operators was **not maintained** then the result would have been:

```
newnum = 18+5-18%5
       = 23-18%5
       = 5%5
       = 0                     ⇒ Which is the wrong answer
```

Note the use of a new include file, viz. the **conio.h** file. It is used for **console input output** functions like **clrscr()** and **getch()** when the Turbo C compiler is used. The first one is used in line-6 to clear the screen i.e. make it blank before something is displayed on it. The second one i.e. **getch()** is used in line-12 to make the output screen wait for the user response (same as the use of **getchar()** for the purpose).

9.8 Operators and Data Types

• Data types and the assignment operator:



Using the assignment operator

Usually the same type of data is used on either side of the assignment operator in an expression. Thus when adding two **floating point** numbers, the result is also assigned to a **float** variable. But in case there are different numeric types on the left and right sides of the assignment operator, **the data type of the final value depends on the data type of variable on the left side of the assignment operator**. The value of the expression is **demoted or promoted** depending on the type of variable on the left hand side of the assignment operator '=' symbol. Therefore:

`int x; float y;`	i.e. **x** is defined as an integer and **y** as a floating point value
`x = 4.9;`	Here **x** will hold the value **4** i.e. it gets **demoted** from a **float** i.e. **4.9** to **4**
`y = 6;`	Here **y** will hold the value **6.000000** i.e. the **int 6** gets **promoted** to a **float 6.000000**

• Data types and the division operator:



Rules for division operator

The result of dividing two values not only depends on the data type of the variable where the result is assigned but also depends on the data types of the operands i.e. the dividend and the divisor.

> **Rule:** When dividing two values if **at least one** of the numbers is a **real** (float or double) type data then the **result is a real type data**. In case **both** the numbers **are of integer type**, the **result** is also an **integer type** data.

The next program piece shows the fact by doing the same calculation on different types of **literal data**.

```
1  int main()
2  {float c;
3   c = 12/5;      printf("%f\n", c);   ⇒ Output is 2.000000 i.e. truncated result
4   c = 12.0/5;    printf("%f\n", c);   ⇒ Output is 2.400000 i.e. correct result
5   c = 12/5.0;    printf("%f\n", c);   ⇒ Output is 2.400000 i.e. correct result
6   c = 12.0/5.0; printf("%f", c);      ⇒ Output is 2.400000 i.e. correct result
7   return 0;
8  }
```

The same rule also applies when **using variables** instead of literals. The following pieces of code display the different combinations of inputs and the resulting outputs depending upon the data types of the variables.



Examples of different types of Division

Case1: Dividend, Divisor, Quotient, all are integers:

```
1  int main()
2  {int a, b, c;
3   printf("Input dividend: "); scanf("%d", &a);
4   printf("Input divisor: "); scanf("%d", &b);
5   c = a/b;
6   printf("The value of %d/%d is %d", a, b, c);
7   return 0;
8  }
```

Output:

```
Input dividend: 4
Input divisor: 8
The value of 4/8 is 0
```

Integer Division: When two integers are divided, the result is an integer.

Case2: Dividend, Divisor integers; Quotient float:

```
1  int main()
2  {int a, b; float c;
3   printf("Input dividend: "); scanf("%d", &a);
4   printf("Input divisor: "); scanf("%d", &b);
5   c = a/b;
6   printf("The value of %d/%d is %f", a, b, c);
7   return 0;
8  }
```

Output:

```
Input dividend: 4
Input divisor: 8
The value of 4/8 is 0.000000
```

Float Division: During division if either the dividend or divisor or both are floats, then the result is also a float.

Case3: Dividend integer: Divisor, Quotient float:

```
1  int main()
2  {int a; float b, c;
3   printf("Input dividend: "); scanf("%d", &a);
4   printf("Input divisor: "); scanf("%f", &b);
5   c = a/b;
6   printf("The value of %d/%f is %f", a, b, c);
7   return 0;
8  }
```

Output:

```
Input dividend: 4
Input divisor: 8
The value of 4/8.000000 is 0.500000
```

Case4: Divisor integer: Dividend, Quotient float:

```
1  int main()
2  {int b; float a, c;
3   printf("Input dividend: "); scanf("%f", &a);
4   printf("Input divisor: "); scanf("%d", &b);
5   c = a/b;
6   printf("The value of %f/%d is %f", a, b, c);
7   return 0;
8  }
```

Output:

```
Input dividend: 4
Input divisor: 8
The value of 4.000000/8 is 0.500000
```

Case5: Dividend, Divisor, Quotient, all are floats:

```
1  int main()
2  {float a, b, c;
3   printf("Input dividend: "); scanf("%f", &a);
4   printf("Input divisor: "); scanf("%f", &b);
5   c = a/b;
6   printf("The value of %f/%f is %f", a, b, c);
7   return 0;
8  }
```

Output:

```
Input dividend: 4
Input divisor: 8
The value of 4.000000/8.000000 is 0.500000
```

In **case-1 and 2** we have got an **unexpected result** as output because the division is basically **integer division** with `a` and `b` both integers. This type of division is called **Integer Division**, which gives the quotient of dividing two integer values. In **case-1** since `c` is also an integer, the output is a pure integer. In **case-2** the result of the integer division is assigned to a floating point variable `c`. Thus the result is still an integer (as the division is carried out first before the assignment) but **expressed as a floating point number** as `c` is a `float`.

In **case-3, 4, and 5** we have got the **expected result** as output because unlike case1 and 2, these divisions are floating point divisions with either `a` or `b` or both as `floats` and the result also assigned to a floating point variable `c`.

Note that in **case-2** we can get a **correct result** using the following technique:

```
5   c = 1.0*a/b;
```

In this modified form, we have multiplied the dividend `a` by the `float` literal `1.0` to make the numerator a floating point value. When the division is performed next, it becomes similar to case-4 i.e. `float/int`, giving the result as a `float`. As the variable `c` has been declared as a `float` in **case-2**, the floating point result will be preserved.

- General rule for data type conversions:

In general when data of different types are present in a given expression, then they are converted to the same type. Two types of type conversions are possible – **implicit and explicit type conversions**. In an implicit type conversion the compiler performs the conversion during the compilation process without any intervention by the programmer.


Data conversion rules

In case of expressions containing **values of a mixed data type**, the **expression is promoted** to the data type of the variable/constant with the largest data type. It follows the following rules during the conversion:

Rules for data conversion

a. If either operand is a long double type data, the other is also converted to a long double type data
b. Else if either operand is a double type data, the other is also converted to a double type data
c. Else if either operand is a float type data, the other is also converted to a float type data
d. Else if an int type data can represent all the values of the original data type (like char, or short int), the value is converted to an int. Else it is converted to an unsigned int. The process is called integral promotion
e. Else if either operand is an unsigned long type data, the other is also converted to an unsigned long type data
f. Else if one of the operands is a long int and the other an unsigned int type data, then if a long int can represent all the values of an unsigned int, the unsigned int is converted to long int. Otherwise both the operands are converted to unsigned long int
g. Else if either operand is a long int type data, the other is also converted to a long int type data
h. Else if either operand is an unsigned type data, the other is also converted to an unsigned type data
i. Else if either operand is an int type data, the other is also converted to an int type data

The following code illustrates the way the type conversion occurs:

```
1  int main()
2  {int a=10, b=6;
3   float c=5.25;
4   double d=8.6, result;
5   result = a * c - a / b + d / c + b;
6   printf("The final result is %lf", result);
7   return 0;
8  }
```

Output:

```
The final result is 59.138095
```

While doing the calculation in line-5, the results of the calculation are type converted in the following way:

`a * c - a / b + d / c + b`	⇒ `int * float`	⇒ result is float = 52.5
`52.5 - a / b + d / c + b`	⇒ `int / int`	⇒ result is int = 1
`52.5 - 1 + d / c + b`	⇒ `double / float`	⇒ result is double = 1.638095
`52.5 - 1 + 1.638095 + b`	⇒ `float - int`	⇒ result is float = 51.5
`51.5 + 1.638095 + b`	⇒ `float + doublet`	⇒ result is double = 53.138095
`53.138095 + b`	⇒ `double + int`	⇒ result is double = 59.138095
`59.138095`	⇒ Final result is a double type data	

Type casting

*In the **explicit form of data type conversion**, the conversion is made a part of the code written. It forces an expression to be of a given data type. The process is known as **Type Casting**. It converts the data of a given data type to another data type for the calculation in this case. The following example shows the method by modifying the code for division used in the last example:*

Type Casting is used to convert data of a given type to another data type.

Modified Case2: Dividend, Divisor integers: Quotient float:

```
1  int main()
2  {int a, b; float c;
3   printf("Input number1: "); scanf("%d", &a);
4   printf("Input number2: "); scanf("%d", &b);
5   c = (float)a/b;
6   printf("The value of %d by %d is %f", a, b, c);
7   return 0;
8  }
```

Output:

```
Input number1: 4
Input number2: 8
The value of 4 by 8 is 0.500000
```

Observe that in **line-5**, we have temporarily converted the variable **a** into a **float** by using the data type identifier **float** written within braces before the variable **a**. In a similar manner we can convert **b** also or **a** and **b** both into a **float** while doing the calculation. However this is a temporary process and after the calculation the variables **a** and **b** will remain as integers. The following lines illustrate the other possibilities.

```
5   c = a/(float)b;
5   c = (float)a/(float)b;
```

9.9 Some worked out examples



Worked out examples

Convert a length entered by the user in inch to cm.

```
1  /*Program-11: inch to cm conversion*/
2  #include<stdio.h>
3  int main()
4   {const float INCHtoCM = 2.54f;
5    float inch, cm;
6    printf("\nInput the length in inches: ");
7    scanf("%f", &inch);
8    cm = INCHtoCM * inch;
9    printf("\nThe required length in cm is = %f", cm);
10   fflush(stdin);
11   getchar();
12   return 0;
13  }
```

Output:

```
Input the length in inches: 4.9
The required length in cm is = 12.446000
```

Find the circumference and surface area of a circle.

```
1  /*Program-12: To find circumference, area of a circle of radius r*/
2   #include<stdio.h>
3   #define PI 3.1416
4   int main()
5   {int r;
6    float circum, area, surf_area, vol;
7    printf("\nInput the Radius (integer number): ");
8    scanf("%d", &r);
9    circum = 2*PI*r;
10   area = PI*r*r;
11   printf("\nRadius=%d, \nCircumference=%f, \nArea=%f", r, circum, area);
12   fflush(stdin);
13   getchar();
14   return 0;
15  }
```

Output:

```
Input the Radius (integer number): 8
Radius=8,
Circumference=50.265598,
Area=201.062393
```

Find the surface area and volume of a sphere of a given area r.

```
/*Program-13: To find surface area and volume of a sphere of radius r*/
#include<stdio.h>
#define PI 3.1416
int main()
{ int r;
  float surf_area, vol;
  printf("\nInput the Radius (integer number): ");
  scanf("%d", &r);
  surf_area = 4*PI*r*r;
  vol = (4.0/3)*PI*r*r*r;
  printf("\nSurface Area = %f, \nVolume = %f", surf_area, vol);
  fflush(stdin);
  getchar();
  return 0;
}
```

Output:

```
Input the Radius (integer number): 6
Surface Area = 452.390411,
Volume = 904.780823
```

To find out the size of the different data types as supported by the compiler that you are using, the following program can be used which uses the **sizeof() function to find the length of the data type** taken as its argument (type the program as it is, compile it and run it to get the result):

```
/*Program-14: To Find the Byte Length of different Data Types*/
#include<stdio.h>
int main()
{ puts("Program to print the size of different Data Types\n");
  printf("\nThe size of an integer data is (in bytes):%d", sizeof(int));
  printf("\nThe size of a short integer data is (in bytes):%d", sizeof(short));
  printf("\nThe size of a long integer data is (in bytes):%d", sizeof(long));
  printf("\nThe size of a float data is (in bytes):%d", sizeof(float));
  printf("\nThe size of a double float data is (in bytes):%d", sizeof(double));
  printf("\nThe size of a char data is (in bytes):%d", sizeof(char));
  fflush(stdin);
  getchar();
  return 0;
}
```

Output:

```
Program to print the size of different Data Types
The size of an integer data is (in bytes):4
The size of a short integer data is (in bytes):2
The size of a long integer data is (in bytes):4
The size of a float data is (in bytes):4
The size of a double float data is (in bytes):8
The size of a char data is (in bytes):1
```

Write a program to find how many hours, minutes, and seconds are there in a given number of seconds.

```
/*Program-15: To find total hour, minute, second in a given number of seconds*/
#include<stdio.h>
#include<conio.h>
```

```
int main()
 { int H, M, S, total, rem;
   clrscr();
   printf("\nInput total number of seconds: ");
   scanf("%d", &total);
   H = total/3600;      /*Integer division gives the integer quotient only for hours*/
   rem = total%3600;    /*Remainder gives the remaining minutes and seconds*/
   M = rem/60;          /*Integer division by 60 on rem gives the number of minutes*/
   S = rem%60;          /*Remainder gives the remaining number of seconds*/
   printf("\n%3d Hour, \n%3d Minute, \n%3d Second", H, M, S);
   fflush(stdin);
   getchar();
   return 0;
 }
```

Output:

```
Input total number of seconds: 11143
  3 Hour,
  5 Minute,
 43 Second
```

Write a program to exchange the values stored in two variables.

```
/*Program-16: To exchange two values using third variable*/
#include<stdio.h>
int main()
 { int num1, num2, temp;
   printf("\nInput first number: ");  scanf("%d", &num1);
   printf("\nInput second number: ");  scanf("%d", &num2);
   temp = num1;         /*Copy the value in num1 in the temporary variable temp*/
   num1 = num2;         /*Copy the value in num2 in the variable num1*/
   num2 = temp;         /*Copy the value stored in temp (i.e. num1) in num2 */
   printf("\nThe exchanged values are: %d, %d", num1, num2);
   fflush(stdin);
   getchar();
   return 0;
 }
```

Output:

```
Input first number: 19
Input second number: 52
The exchanged values are: 52, 19
```

C to it that .....

The following examples demonstrate the areas where mistakes are common. The whole program is not written always and only the relevant parts are shown.

Find the area of a circle of radius r.

```
#include <stdio.h>
#define PI 3.1416;
int main()
 {float r, area;
  printf("\nEnter the radius of the circle: "); scanf("%f", &r);
  area = PI*r*r;
  printf("\nArea of circle is %f", area);
  return 0;
 }
```

At a first glance it may seem that the program is ok. But the moment you will try to compile it (i.e. converting the source code to machine code), it will not compile and will give an error message. The error lies in the **use of the statement terminator** at the end of line-2. Remember that:

No pre-processor directive should end with a semicolon ';'

Rectify the line as follows and the program should immediately compile:

```
2   #define PI 3.1416
```

Find the area of a rectangle.

```
1   #include <stdio.h>
2   int main();
3    {int len, br, area;
4     printf("\nEnter the length of the rectangle: "); scanf("%d", &len);
5     printf("\nEnter the breadth of the rectangle: "); scanf("%d", &br);
6     area = len*br;
7     printf("\nArea of rectangle is %d", area);
8     return 0;
9    }
```

When you try to compile the above program a **list of error messages will appear**. The reason is again the **wrong use of statement terminator**. There will not be any statement terminator after the name of the `main()` function in the function header in line-2.

```
2   main()
```

Write a program to convert the length in inches to feet.

```
1   #include <stdio.h>
2   int main()
3    {int inches; float feet;
4     printf("\nEnter the length in inches: "); scanf("%d", &inches);
5     feet = inches/12;
6     printf("\nThe required feet = %f", feet);
7     return 0;
8    }
```

The above program has a flaw in line-5. The division should be a floating point division to get the exact result. Either you will have to use a floating point variable for inches or you will have to use typecasting. In this program there is another way out. Since the denominator is a literal, to make the program run we can convert it to a floating point value as shown below:

```
5     feet = inches/12.0;
```

Find the volume of a sphere of radius r.

```
1   #include <stdio.h>
2   #define PI 3.1416
3   main()
4    {float r, vol;
5     printf("\nEnter the radius of the sphere: ");
6     scanf("%d", &r);
7     vol = 4.0/3*PI*r*r*r;
8     printf("\nVolume of the sphere is %d", vol);
9     return (0);
10   }
```

Output:

```
Enter the radius of the sphere: 10
Volume of the sphere is 0
```

When you compile the above program, the compilation will show no error messages. But when you run the program **you will get total garbage output**. The fault lies in the **incorrect use of the format specifiers** in line-6 and line-8. We have declared the two variables `r` and `vol` as floating point numbers. Hence instead of the `%d` format specifier we should use the `%f` format specifier. The compiler will wrongly interpret the data if integer format specifiers are used with a floating point type data. The corrected lines will look like this:

```
6   scanf("%f", &r);

8   printf("\nVolume of the sphere is %f", vol);
```

Rectified Program Output:

```
Enter the radius of the sphere: 10
Volume of the sphere is 4188.799805
```

Program to input a number and find its square.

```
1  #include <stdio.h>
2  main()
3   {int num, sqr;
4    printf("\nEnter number to find square: ");
5    scanf("%d", num);
6    sqr = num * num;
7    printf("\nThe required square is %d", &sqr);
8    return (0);
9   }
```

Output:

```
Enter number to find square: 5
The required square is 1329603584
```

The reason for this incorrect output is the wrong use of the `scanf()` function in line-5 and `printf()` function in line-7. Since the variable `num` is an integer, **there should be the address operator** '`&`' before the variable name in `scanf()`. Moreover there should not be any address operator before the variable name in the `printf()` function. The rectified statements are shown below:

```
5   scanf("%d", &num);

7   printf("\nThe required square is %d", sqr);
```

Rectified Program Output:

```
Enter number to find square: 5
The required square is 25
```

*****Whenever you have some **garbage or unpredicted output check the following*****

CAUTION!

- Whether you have used the **format specifiers correctly** in the `scanf()` and `printf()` functions.
- Whether you have used the format specifiers **in correct order** in the `scanf()` and `printf()` functions when there are **multiple format specifiers**.
- Whether you have **not used the address operator** along with the variable in a `scanf()` operation as `scanf("%d", x)`, instead of `scanf("%d", &x)` for numeric and character variables.
- Whether you have by mistake used the address operator along with a variable in the data list of a `printf()` statement as `printf("\nArea = %f", &area)` instead of `printf("\nArea = %f", area)`.

The Fact File

- **Input and output can be from standard input/output devices like keyboard, VDU or can be from files**
- The most important of these output functions in C is the versatile printf( ), which can display all types of data and can work with multiple arguments in one single parameter. In addition, printf( ) can also format the way the data appears. Example: printf("The value of area = %d", x);
- **Any number of data can be passed to the printf( ) function, but a format specifier must be included for each item and moreover the values in the data list must appear in the same order as their specifiers.**
- The format specifiers are used for formatting the way the data is displayed. This is done using 'field width specifiers'
- **Format specifiers used include: %d for int, %f for float, %c for char, %lf for double, %ld for long etc.**
- The general form of a format specifier is %N.n, where N is the width of the data (including the decimal point for floats) and n indicates the precision or the number of digits/characters displayed
- **Apart from containing the format specifiers for formatting the text, the printf( ) function can also control the way the cursor moves on the screen by using special codes called Escape Sequences. Each escape sequence begins with a "\" which identifies the character that follows as an escape sequence**
- Escape sequence \n = newline, \t = tab, \r = carriage return, \b = backspace, \\ = \, \" = ", \' = '
- **scanf( ) is the general purpose input function which can be used to input all types of data from the keyboard. Scanf( ) scans formatted data from the keyboard. Example: scanf("%d", &x);**
- in the scanf function, the address of a variable is denoted by placing the address operator '&' before the variable name for numeric (i.e. int, float, double etc.) and character (i.e. char) type variables. The address operator points to the memory address at which the contents of the variable are stored
- **Be careful in specifying the order in which the conversion characters and the variable names are written. If the order does not match, then proper data will not be input into the variables.**
- The putchar( ) function displays a single character value on the monitor. The parameter of the putchar( ) function must be either a char literal, a char constant or a char variable. Example: putchar(ch);
- **Whereas the putchar( ) function displays a character on the screen, the puts( ) function is used to display a text string on the screen. Similar to putchar( ), the puts( ) function displays unformatted string starting from a new line**
- Always use the fflush(stdin) statement before entering a character type or string type data using the scanf( ), getchar( ) or gets( ) functions to avoid any malfunction.
- **Similar to the putchar( ) function which is used to output an unformatted character data, there is the getchar( ) function to input any unformatted character data. Example: ch=getchar( );**
- Apart from the scanf( ) function there is the gets( ) function to input any type of unformatted text/string into a variable. The gets( ) function can be used to input any type of string data. Example: gets(str);
- **The assignment operator is used to assign the result of a calculation or a value from the right of the operator to the left. All calculations should be done on the right hand side of the assignment operator. (This is different from the equality operator).**
- Operators with a higher precedence are evaluated before operators with a lower precedence in an expression. When a group of operators of the same precedence are present in an expression, then associativity indicates the direction in which the operators will be evaluated.
- **When there is a mismatch of variable types during an assignment, then the value of the expression is demoted or promoted depending on the data type of the variable on the left hand side of the = sign**
- When dividing two literal values or two variables and assigning the result to a float, at least one of the numbers must be of real type if the correct result is desired as a floating point value
- **Based on their mode of operation, operators can be of different types viz. arithmetic, assignment, relational, increment, & logical**
- The assignment operator is used to assign the result of a calculation or a value from the right of the operator to the left. All calculations should be done on the right hand side of the assignment operator
- **When dividing two values if at least one value is a real (float or double) type data then the result is a real type data. In case both the numbers are of integer type, the result is also an integer type data**
- In general when data of different types are present in a given expression, then they are converted to the same type. Two types of type conversions are possible – implicit and explicit type conversions
- **In an implicit type conversion the compiler performs the conversion during the compilation process without any intervention by the programmer**
- In case of expressions containing values of a mixed data type, the expression is promoted to the data type of the variable/constant with the largest data type

- **In the explicit form of data type conversion, the conversion is made a part of the code written. It forces an expression to be of a given data type. The process is known as Type Casting**

Review Questions

Q1. Multiple Choice Questions. Select from any one of the four options. **1 each**

i) The first argument of the printf() function is called the:
a. argument list b. data list c. control string d. argument string

ii) The format specifiers function as _____ for displaying data in printf() function:
a. placeholders b. literals c. variables d. constants

iii) The format specifier for a long signed integer in C is:
a. %L b. %Ld c. %ld d. %lf

iv) The format specifier for a long double type data in C is:
a. %D b. %ld c. %lf d. %Lf

v) Which of the following format specifiers does not represent a real type data in C?
a. %g b. %e c. %d d. %f

vi) There can be how many control strings in a given printf() function call?
a. 3 b. many c. 2 d. 1

vii) Which input function in C can be used to input all types of data?
a. getchar() b. scanf() c. gets() d. getch()

viii) Format specifiers used in the scanf() function are also called:
a. control characters b. display characters c. conversion characters d. data characters

ix) Which of the following is not a white space character in C?
a. enter b. blank c. tab d. comma

x) Which function is used to clear the input buffer in C?
a. clrscr() b. fflush() c. cls() d. empty()

xi) What is the argument of the fflush function to clear the input buffer in C?
a. stdin b. stdout c. stdio d. stdlib

xii) For %9.3f, how many places are reserved for the integer part and how many for the decimal part?
a. 5, 4 b. 6, 3 c. 5, 3 d. 9, 3

xiii) What output will be shown by the code: printf("**%8.5s", "cricket);
a. ** cri b. ** cric c. ** crick d. ** cricket

xiv) Which escape sequence represents the back space character in C?
a. \n b. \r c. \s d. \b

xv) Which of the following is not an escape sequence in C?
a. \r b. \d c. \\ d. \"

xvi) Which escape sequence can move the cursor to the beginning of the same line?
a. \L b. \b c. \r d. \n

xvii) Which function can only display a single character value on the screen?
a. putchar() b. putcharacter() c. putch() d. puts()

xviii) Which function can be used only to display a text string on the screen?
a. putch() b. putcharacter() c. puts() d. putchar()

xix) Which function can be used to input an unformatted character data in C?
a. scanf() b. printf() c. gets() d. getchar()

xx) Which function can be used to input an unformatted string data in C?
a. gets() b. getchar() c. scanf() d. printf()

xxi) If a=34, b=9, then a%b will be equal to ___:
a. 9 b. 3 c. 7 d. 4

xxii) The process of converting a given data type to some other type is called:
a. type checking b. type converting c. type defining d. type casting

xxiii) Which of the following calculations will not produce the exact result of the division?
a. 7.0/4 b. 7/4 c. 7/4.0 d. 7.0/4.0

Q2. **Short Answer type questions:** **1 each**

i) Name any two functions used in C for unformatted data output.
ii) Name any two functions used in C for unformatted data input.
iii) State one use of a format specifier.
iv) Write any two escape sequences that are white space characters also.
v) What are the format specifiers used for a long double type and a string type data?
vi) Name the two types of divisions possible in C.
vii) What is the meaning of the term 'associativity' with respect to arithmetic operators in C?
viii) Write the operators +, /, *, - in order of precedence in C.
ix) What is typecasting?
x) State one use of the getchar() function in C.
xi) What is the meaning of the format specifier %-10d?
xii) What is the difference between the / and the % arithmetic operators in C?

Q3. **Long Answer type questions:** **7 each**

i) Explain the working of the printf() function with respect to its arguments. What is the difference between the printf() and puts() functions? What is the use of fflush() function? Name the format specifier for a unsigned int type data. 2+2+2+1

ii) What do you mean byt operator precedence and associativity in C? What are escape sequences? Name any two escape sequences and mention their uses. What is type casting? 2+2+2+1

iii) Explain the difference between integer and floating point division in C with the help of proper examples. What do you mean by white space characters? Name one operator with right to left associativity in C. 4+2+1

Q4. **Assignment Programs:** **4 each**

i) Write a C program to convert a temperature value input in Fahrenheit to Celsius scale using the relation C = 5(F-32)/9

ii) Write a C program to find the area of a square when the user inputs the length of the diagonal.

iii) Input s (distance), u (initial velocity), t (time) and find the acceleration given by: $f = 2(s - u\,t)/t^2$

iv) Input the base radius and height of a solid cone and find its total surface area and volume.

v) Input the radius r of a circle which is perfectly inscribed inside a square as shown in the adjacent figure. Find the area of the dark shaded portion of the square.



vi) You have 100gm, 50gm, 20gm, 10gm, 5 gm, 2gm, and 1gm weights available with you. Input a given weight in grams and find out the number and type of each weight that are required to balance the weight in a common balance. [Hint: Try using integer division and remainder operator]

vii) Write a C program to find the cost of white washing a room. The user inputs the length, breadth and height of the room and the cost per unit area of whitewashing.

viii) Write a C program that inputs four digits and converts them into a single decimal integer. [i.e. if the digits input are say 2,8,4 & 9 then the program will give a single number output as 2849. Do not write the 4 digits side by side. Try to use the minimum of variables to do this.]

ix) The astronomical unit is defined as the average distance from Earth to Sun, the value of which is 149,597,870 km. Write a C program that converts the distance in light years input by the user to distance in astronomical units. Test the program with 4.2 light years, the distance of our nearest star. Take the variables as of type double. The speed of light is 300,000 km/sec.

x) Write a C program to display the Octal and Hexadecimal equivalent of a Decimal number. [Hint: Use the format specifiers for Octal and Hexadecimal numbers].

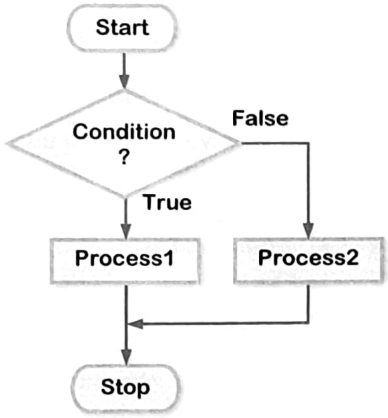
xi) Write a C program to interchange the contents of two variables a & b without using a third variable. [Hint: Input the given numbers a & b: assign **a + b to a i.e. a ← a + b**
assign **a – b** to **b** i.e. **b ← a – b**
assign **a – b** to **a** i.e. **a ← a – b**]

CHAPTER 10
Decision Making and Branching in C

- **Introduction** 10-1
- **The Relational Operators** 10-1
- **The if-else statement** 10-2
- **The Logical Operators (AND, OR, NOT)** 10-10
- **The Conditional Operator (? :)** 10-16
- **The switch-case-default structure** 10-19
- **Some worked out examples** 10-22

10.1 Introduction

All the programs that we did so far ran in a linear manner i.e. all the lines written in the program were executed when the program was run. However, there may be cases where **depending upon one or more conditions,** some parts of the code may get executed and some parts may not. Accordingly there **can be more than one output**.



Flowchart for decision making

In a program whenever there are **more than one possible outputs** depending upon some condition(s) for a particular set of input(s), the situation is called program **branching**. The program control branches to different segments executing different sets of codes for each possible situation as shown by the flowchart.

A logical condition evaluates to either True or False

Depending upon whether the condition is True (Yes) or False (No) the program branches to **process1** or **process2**. Thus when the condition is true **process1** gets executed and the code for **process2** is not executed. Similarly when the condition is false, **process2** gets executed and the code for **process1** is not executed. More than one branching is also possible. Such a situation where branching takes place is handled by C using different constructs like **if, if-else, conditional operator (? :)**, and **switch-case**. Each of these is discussed in the following sections.

10.2 The Relational Operators

To test or verify the conditions based on which branching takes place, C provides us with some **special operators** along with the conventional ones as given below. These special operators called **relational operators** allow us to **compare two values** to take a decision.

Operator	Example	Function
Less Than **<**	**x<y**	The expression evaluates to **True or 1** if **x is less than y** else it evaluates to False or 0. Thus for **x=3** and **y=5**, **x<y** evaluates to True or 1, while for **x=5** and **y=4**, **x<y** evaluates to False or 0.
Less Than or Equal **<=**	**x<=y**	The expression evaluates to **True or 1** if **x is less than or equal to y** else it evaluates to False or 0. Thus for **x=3** and **y=5**, **x<=y** evaluates to True or 1. Also for **x=5** and **y=5**, **x<=y** evaluates to True or 1.
Greater Than **>**	**x>y**	The expression evaluates to **True or 1** if **x is greater than y** else it evaluates to False or 0. Thus for **x=5** and **y=3**, **x>y** evaluates to True or 1, while for **x=4** and **y=5**, **x>y** evaluates to False or 0.
Greater Than or Equal **>=**	**x>=y**	The expression evaluates to **True or 1** if **x is greater than or equal to y** else it evaluates to False or 0. Thus for **x=5** and **y=3**, **x>=y** evaluates to True or 1. Also for **x=5** and **y=5**, **x>=y** evaluates to True or 1.
Equality **==**	**x==y**	The expression evaluates to **True or 1** if **x and y have the same value**. Do not confuse this with the assignment operator (=), where x=y implies the value of y is assigned to x. Thus if x=3 and y=5 then x==y is False or 0 since x and y are not equal, while x=y results in x been assigned the value of y i.e. 5 and x becomes equal to 5. On the other hand after the operation x==y, x and y retain their old values of 3 and 5 respectively.
Not Equal **!=**	**x!=y**	The expression evaluates to True or 1 if x and y do not have the same value. Thus for x=3 and y=5, x!=y evaluates to True or 1, while for x=4 and y=4, x!=y evaluates to False or 0.

Relational Operators

The equality operator == compares two values to test if they are same

10.3 The if-else statement

In C, the `if` statement is used to take a decision. The `if` statement tells the compiler that the instructions to follow are based on a decision. The general structure of the `if-else` control statement is given below:

Structure of if-else statement

```
If (condition_is_true)
   { execute_statement_set1;
     … … …;
   }
else
   { execute_statement_set2;
     … … …;
   }
```

No semicolon after the condition

If the condition within the braces is True then execute the set of statements within these curly braces { }, also called the *if* block statements

Else if the condition is False then execute the set of statements within these curly braces { }, also called the *else* block statements

The if block and else block statements should be enclosed within { } braces

The condition to check is placed inside braces after the `if` keyword. If the condition is True then the block of **statement set1** within curly braces under `if` gets executed. Else, if the condition is False then the block of **statement set2** within curly braces under `else` gets executed. Statement sets 1 & 2 can consist of one or more statements as per program requirement. Please note that there is **no semicolon (;) after the braces enclosing the condition**. The semicolon comes after the first statement to execute if the condition is True.

The program given below inputs the total purchase amount of a customer and **calculates the net amount payable** after offering different discount values depending upon the total amount.

Program to show use of if-else

```
1   /*Program-17: Program to calculate different discounts using if-else*/
2   #include<stdio.h>
3   int main()
4   {float amount, total;
5    printf("\nEnter the total purchase amount (Rs.): ");
6    scanf("%f", &amount);
7    if( amount < 3000.0 )
8       {total = 0.9*amount;
9        printf("\n10 percent discount is provided");
10      }
11   else
12      {total = 0.8*amount;
13       printf("\n20 percent discount is provided");
14      }
15   printf("\nThe net amount payable = Rs. %0.2f", total);
16   return 0;
17  }
```

Output1:

```
Enter the total purchase amount (Rs.): 1500
10 percent discount is provided
The net amount payable = Rs. 1350.00
```

Output2:

```
Enter the total purchase amount (Rs.): 4625
20 percent discount is provided
The net amount payable = Rs. 4162.50
```

Depending upon the amount, if the amount is less than `Rs.3000/-` then the condition in line-7 gets true and a 10% discount is calculated in line-8. The discount offerred is then printed in line-9. If the purchase value is more than or equal to `Rs.3000/-` then the condition in line-7 becomes false and the `else` statement gets executed in line-11. In that case the discount provided is 20% and the discount offerred is printed in line-13.

The discounted amount to pay is printed in line-15 outside the `if-else` block. Note that if the condition is True, then the `if` block statements get executed from line-8 to 10 and then the program control goes directly to line-15 to print the discounted amount.

However there are **several variations** of the above construct. The variations are discussed below.

if without *else* statement:

In case **there are no statements to execute if the condition is False**, then the `else` **statement is dropped**. Only the statements inside the curly braces under `if` get executed in case the condition is true. Else, the control passes to the line immediately after the end of the curly braces of the `if` block.

If without else statement

```
If (condition is true)
  { execute statement set;
    ... ... ... ...;
  }
```

Program control goes to this point skipping the statement set if the condition is NOT true

The program below inputs a number and prints the absolute value of the number.

You can have an if statement without an else statement

```
1  /*Program-18: Use of only if statement*/
2  #include<stdio.h>
3  int main()
4  {float num;
5   printf("\nEnter any integer: ");
6   scanf("%f", &num);
7   if(num < 0)                    /*Condition checked using if statement*/
8     {num = num*(-1);}            /*Statement that executes if the condition is True*/
9   printf("\nThe absolute value is %f", num);
10  return 0;
11 }
```

Output1:

```
Enter any integer: -79.5
The absolute value is 79.5
```

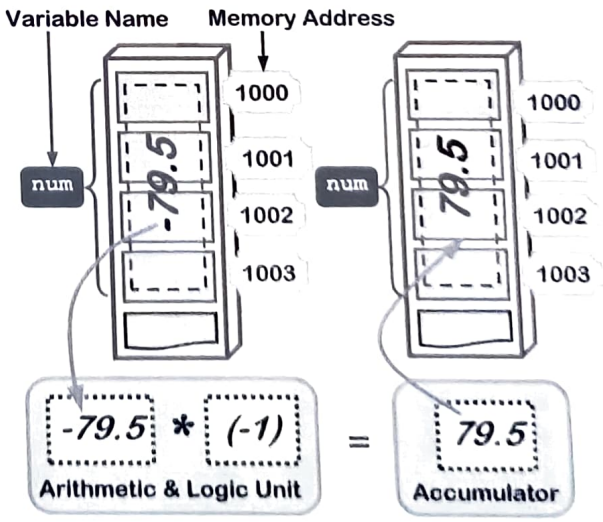
Output2:

```
Enter any integer: 45.4
The absolute value is 45.4
```

In line-7 the condition (`num < 0`) that checks if the input number `num` is negative or not, is placed within braces after the `if` keyword. If the condition is True (for a negative number), the statement in line-8 is executed. In that case in line-8 the value `num` is multiplied by (-1) to make it positive and again stored back in `num`. On the other hand if the condition is False (for a positive number) then there is no need to change the sign of the number and the statement in line-8 will not get executed. In that case the value stored in `num` will remain same. The final result showing the absolute value of the entered number is printed in line-9.

Before we proceed further, let us clearly understand the meaning of the statement `num=num*(-1)` in line-8.

Apparently this may seem to be an absurd statement. However in a programming language the process is a completely valid one. The memory diagram on the right illustrates the meaning of the above statement.



As shown in the first output, when the value `-79.5` is input in line-6, it gets stored in the memory location 1000 allocated for the variable `num` as shown in the diagram. In line-8 when the program encounters the statement `num=num*(-1)`, it fetches from memory the value stored in the variable `num`, to perform the calculation on the right side of the assignment operator.

The value `-79.5` is then multiplied with `(-1)` in the ALU portion of the CPU and the result is temporarily stored in the memory register called Accumulator. From the Accumulator, the result is then assigned back to the memory location for `num` (remember that the calculated

result at the right side of the assignment operator is assigned to the variable at the left side of the assignment operator).

The new value `79.5` from the Accumulator will now overwrite the old value in `num` i.e. `-79.5`. So the final content of the memory after the execution of the statement will be `79.5`.

Output1 shows what happens when someone enters a negative number like `-79.5`. The program as usual prints `The absolute value is 79.5`. In **Output2** we find that when `45.4` is entered, the condition `num<0` is not satisfied and the number is not multiplied by (-1). Finally the program prints the positive value as usual as `The absolute value is 45.4`.



Single statement execution

- **Single statement to execute if condition is True or False:**

If there is **only one statement to execute** in case the `if` condition is True (or False), **the curly braces** enclosing the `if` block or the `else` block are **optional**. The statement to execute can be put beside the condition or in the next line, without the curly braces.

This is because in case the condition is true, then the statement immediate to the condition is always executed by default. Only to execute **multiple statements** in the `if` or `else` block we put them inside a pair of curly braces.

A single statement after if or else need not be enclosed within { } braces

```
If (condition is true)
  execute_single_statement_by_default;
```

Curly braces { } not required in case we have a single statement to execute when the case condition is true

The program below inputs a number and prints if it is even or odd.

```
1  /*Program-19: To check if even or odd number with single statement under if and else */
2  #include<stdio.h>
3  int main()
4  {int num;
5   printf("\nEnter any integer: ");
6   scanf("%d", &num);
7   if( num%2 == 0 )                    /*Condition checks if number is divisible by 2*/
8     printf("\nNumber is even");       /*If divisible remainder is 0 and condition is True*/
9   else
10    printf("\nNumber is odd");        /*If not divisible then condition is False*/
11  return 0;
12 }
```

Output1:

```
Enter any integer: 6
Number is even
```

In the above program, line-7 checks if the input number `num` is divisible by 2 or not. If divisible, then remainder of division given by `num%2` will be `0` (zero) and the condition will be True when compared with 0 using the equality operator. In that case the program statement in line-8 prints "`Number is even`".

If the number is odd, then the remainder of the division in line-7 will be 1, and the condition will be False. In that case the program statement in line-10 after the `else` keyword prints "Number is odd".

The above example shows how a single statement is written after the `if` and `else` statements without the use of the curly braces `{}`.

A program can have multiple if statements but without an else statement

The next program checks **if a number is negative, positive or zero**. It is an example of a program with only `if` statements and single statements to execute if the condition is true.

```
1  /*Program-20: Use of multiple single if statements*/
2  #include<stdio.h>
3  int main()
4  {int num;
5   printf("\nEnter any integer: ");
6   scanf("%d", &num);
```

```
7   if(num<0) printf("\nNumber is negative");  /*Print statement in same line as if*/
8   if(num==0) printf("\nNumber is zero");     /*Print statement in same line as if*/
9   if(num>0) printf("\nNumber is positive");  /*Print statement in same line as if*/
10  return 0;
11 }
```

Output1:

```
Enter any integer: 50
Number is positive
```

Output2:

```
Enter any integer: -20
Number is negative
```

Output3:

```
Enter any integer: 0
Number is zero
```

In the above program, there are **three** `if` statements **without any** `else` **statement**. Each `if` statement checks a particular condition and if it is found true, then the corresponding single `printf()` statement gets executed. Note that as there is a single statement to execute, hence we have not enclosed them within curly braces.

Line-7 checks if the number is less than '0' by the condition `num<0`. If it is found true then the following `printf()` statement prints "`Number is negative`". Line-8 checks if the number is zero by using the condition `num==0`. If it is found true then the following `printf()` statement prints "`Number is zero`". Finally in case the number is positive, then the `if` statement in line-9 gets true and the corresponding `printf()` function prints "`Number is positive`".

Note that though there are 3 `if` statements, but **only any one of them finally gets executed** depending upon which condition is True. One more thing to note is the use of the equality operator (==) in line-8. It is used to compare the value stored in the variable `num` with the value '0'. **In case these are same, then only the condition gets true.**

- Nesting of *if-else* statements:

Sometimes you may be required to put an `if-else` statement within the `if`-block or `else`-block of another `if-else` statement. Such a process is known as **nesting of** `if-else` **statements**. To show how nesting works, we have written a modified version of Program-20 below.

Nesting of if-els statements

An if-else statement can put inside another if or e block. This is called Nesting if-else

```
1  /*Program-21: Use of nested if-else statement*/
2  #include<stdio.h>
3  int main()
4  {int num;
5   printf("\nEnter any integer: ");
6   scanf("%d", &num);
7   if(num<0)                              /*Outer if statement*/
8     printf("\nNumber is negative");
9   else                                   /*Outer else statement*/
10    {  if(num>0)                         /*Inner if statement within outer else*/
11         printf("\nNumber is positive");
12       else                             /*Inner else statement within outer else*/
13         printf("\nNumber is zero");
14    }
15  return(0);
16 }
```

Output:

```
Enter any integer: 32
Number is positive
```

In this program we have used **one main `if-else` block**. Within the `else` block from line-10 we have placed another `if-else` block. This **placing of one `if-else` block within another** is known as **nesting**.

In case the condition `num<0` in line-7 is not satisfied then the `else` statement in line-9 gets executed. All the statements from line-10 to line-14 are under the outer `else` statement in line-9. Hence these are put within a pair of curly braces between line-10 and 14. These curly braces indicate that in case the `if` condition is False then as part of the `else` block in line-9, all the statements from line-10 to line-13 should be executed.

The first `if` statement in line-7 checks if the number is negative. If so, it prints the same in line-8. If not, the `else` statement of line-9 is executed. (Note that if a number is not negative, then it can be either positive or zero, and this is then tested within the outer `else`).

Within the `else` statement in line-10 it is first checked if `num>0`. In case the condition is true the `printf()` in line-11 prints `Number is positive`. In case the condition is false, the `else` statement in line-12 gets executed and the `printf()` in line-13 prints `Number is zero` as this is the only option left in case the number is neither negative nor positive.

Line-7 to 14 shown below highlights the scope of the two `if-else` blocks. The `else` in line-9 goes with the `if` in line-7 as **indicated by the outer dotted line** and the `else` in line-12 goes with the `if` in line-10.

Outer if-else block

Inner if-else block within the outer else block

```
7   if(num<0)
8      printf("\nNumber is negative");
9   else
10     {  if(num>0)
11           printf("\nNumber is positive");
12         else
13           printf("\nNumber is zero");
14     }
```

Note how the lines under the `if` and `else` keywords are **indented to the right** while writing the program. Though this is not essential but **it makes reading and debugging the program much easier**. Hence always try to follow this style while writing your own programs.

The above example shows us how multiple statements can be executed within one conditional block and the **concept of nesting one block of code within another**. The next program shows nesting used in both the `if` and the `else` blocks.

The following program checks if a number is even or odd. In case the number is even, it checks if it ends with 6 or not. On the other hand, if the number is odd, it checks if the number ends with 7 or not.



if-else statement within if and else

```
1  /*Program-22: Use of nested if-else statement*/
2  #include<stdio.h>
3  int main()
4  {int num;
5   printf("\nEnter any integer: ");
6   scanf("%d", &num);
7   if(num%2==0)                          /*Outer if statement*/
8     {  if(num%10==6)                    /*Inner if statement within outer if*/
9          printf("\nNumber is even and ends with 6");
10       else                            /*Inner else statement within outer if*/
11         printf("\nNumber is even, but does not end with 6");
12    }
13  else                                  /*Outer else statement*/
14    {  if(num%10==7)                    /*Inner if statement within outer else*/
15         printf("\nNumber is odd and ends with 7");
16       else                            /*Inner else statement within outer else*/
17         printf("\nNumber is odd, but does not end with 7");
18    }
19  return 0;
20 }
```

Output1:

```
Enter any integer: 56
Number is even and ends with 6
```

Output2:

```
Enter any integer: 39
Number is odd, but does not end with 7
```

Line-7 and line-13 represent the outer **if-else** block. The outer **if** checks if the number is even. If so, the **if** block statements from line-8 to line-12 get executed. Within this block, the inner **if** statement in line-8 checks if the number ends with 6 by using the remainder of dividing the number by 10 (e.g. 56%10 is equal to 6). If so, it prints the required message. If not so, then it executes the **else** statement of line-10 and prints the required message in line-11.

If the number is odd, then the condition in the outer **if** block is false, and the outer **else** statement in line-13 gets executed. Within this block, the inner **if** statement in line-14 checks if the number ends with 7 by using the remainder of dividing the number by 10 (e.g. 27%10 is equal to 7). If so, it prints the required message. If not so, then it executes the inner **else** statement of line-16 and prints the required message in line-17.

- The *if-else* ladder:



The if-else ladder

Certain situations may require multiple nesting i.e. putting several blocks of **if-else** one within the other. The program inputs the number of sides for a given closed shape and depending upon the number input, prints the name of the probable shape.

```
1  /*Program-23: Use of multiple if-else nesting*/
2  #include<stdio.h>
3  int main()
4  {int side;
5   printf("\nEnter the number of sides: ");
6   scanf("%d", &side);
7   if(side==1)
8      printf("\nFigure can be a circle");
9  else
10   {  if(side==2)
11        printf("\nFigure with 2 sides");
12     else
13       {  if(side==3)
14             printf("\nFigure can be a triangle");
15          else
16            {  if(side==4)
17                  printf("\nFigure can be a quadrilateral");
18               else
19                 {  if(side==5)
20                       printf("\nFigure can be a pentagon");
21                    else
22                      {   if(side==6)
23                            printf("\nFigure can be a hexagon");
24                          else
25                            printf("\nFigure with sides more than 6");
26                      }
27                 }
28            }
29       }
30   }
31 return 0;
32 }
```

In the above program we have placed an **if-else** block within each **else** block (excepting the last) to get multiple nesting. If the first condition is True, then it prints the statement in line-8. Else, if False, the control enters the **else** block in line-10. Within this **else** block, it checks the second condition in line-10. If True, it prints the statement in line-11. If this is also False, then the control enters the **else** block of line-13.

The process continues in this manner till the last condition is checked for number of sides equal to 6 in line-22. If True, it prints the statement in line-23. Else the program stops any further checking and prints the final message "Figure with sides more than 6", when all the other conditions become False.

An if-else ladder represents multiple nested if-else statements

The **else** blocks are marked by dotted lines in the program listing. Each dotted line points to the opening and closing braces of a given **else** block.

Depending upon which condition is True, only one of the **printf()** statements will get executed always. The moment a condition becomes True, the program stops checking any further conditions and after executing the statement under the True condition, comes out of the nested **if-else** structure to line-31.

Note the following points in the above program:

- Only a **single statement is executed** when any condition becomes True under an **if** statement
- No **if** block has a nested **if-else** block within it
- Only an **else** block has nested **if-else** block within it (except the last or innermost **else**)

Under such a situation you can do away with all the curly braces and indentations and write a much simpler version of the above program in a style called the **if-else ladder**, as shown below.

If-else ladder program

```
1  /*Program-23a: Use of if-else ladder structure*/
2  #include<stdio.h>
3  int main()
4  {int side;
5   printf("\nEnter the number of sides: ");
6   scanf("%d", &side);
7   if(side==1)
8      printf("\nFigure can be a circle");
9   else if(side==2)
10     printf("\nFigure with 2 sides");
11  else if(side==3)
12     printf("\nFigure can be a triangle");
13  else if(side==4)
14     printf("\nFigure can be a quadrilateral");
15  else if(side==5)
16     printf("\nFigure can be a pentagon");
17  else if(side==6)
18     printf("\nFigure can be a hexagon");
19  else
20     printf("\nFigure with sides more than 6");
21  return 0;
22 }
```

Output:

```
Enter the number of sides: 4
Figure can be a quadrilateral
```

Since each **if** statement has to execute a single statement in case it is true, the curly braces {} are not required for the **printf()** statements. Moreover the next **if** after an **else** option is written in the same line as the **else**. The structure is **called a ladder** as the shape of the program from line-7 to 20 looks like the rungs of a ladder.

Conditions without using relational operators

- **Conditions without using relational operators:**

We have seen how relational operators are used to check conditions in an **if** statement. Each condition either evaluates to True or False. C automatically assigns the numeric value '1' for a True condition and '0' for a

False condition. In general **any expression which evaluates to a non-zero positive or negative value** (e.g. 1, 230, 3.92, –210, –8.3 etc.) **is taken as True** or 1 and **only a zero value is taken as False** or 0.

Hence for two values `a=6` and `b=9` the `if` conditions in the following examples will evaluate to:

- `if(a)` ⇒ Condition is **true** or equals to 1 as `a=6=(non-zero positive)=True=1`
- `if(-a)` ⇒ Condition is **true** or equals to 1 as `-a=-6=(non-zero negative)=True=1`
- `if(1)` ⇒ Condition is **true** as the constant value 1 is taken as True
- `if(0)` ⇒ Condition is **false** as the constant value 0 is taken as False
- `if(b-a)` ⇒ Condition is **true** or equals to 1 as `(b-a)=(3)=(non-zero positive)=True=1`
- `if(-(b-a))` ⇒ Condition is **true** or equals to 1 as `(-(b-a))=(-3)=(non-zero negative)=True=1`
- `if(a*3-b*2)` ⇒ Condition is **false** or equals to 0 as `(a*3-b*2)=(18-18)=(zero)=False=0`

The following program, that uses an if-else ladder structure, evaluates multiple conditions without using relational operators. It inputs a number (up to 4 digits) and prints the number of digits present in it.

```
1  /*Program-24: Use of conditions without relational operators*/
2  include<stdio.h>
3  int main()
4  {int num;
5   printf("\nEnter an integer value (<5 digits): ");
6   scanf("%d", &num);
7   if(num/1000)
8       printf("\nA 4 digit number");
9   else  if(num/100)
10      printf("\nA 3 digit number");
11  else  if(num/10)
12      printf("\nA 2 digit number");
13  else
14      printf("\nA 1 digit number");
15  return 0;
16 }
```

Output:

```
Enter an integer value (<5 digits): 34
A 2 digit number
```

Let us now discuss the logic we have used to count the number of digits. We have taken the input number `num` as an **integer**. Thus:

- If `num = 2456`: (4 digits) `2456/1000` (integer division) evaluates to 2 which is a **non-zero** number and taken as ***True***. Thus line-7 becomes true and prints "`A 4 digit number`"
- If `num = 169`: (3 digits) `169/1000` evaluates to 0 (integer division) in line-7, and is taken as ***false***. So the `else` statement in line-9 gets executed. Here `169/100` evaluates to 1, which is a non-zero number and taken as ***true*** and line-10 prints "`A 3 digit number`"
- If `num = 34`: (2 digits) `34/1000` evaluates to 0 (integer division) in line-7, and is taken as ***false*** so the `else` statement in line-9 gets executed. Here `34/100` also equals to 0 and is taken as ***false*** so control passes to the `else` in line-11. Here `34/10` evaluates to 3 which is non-zero and taken as ***true***. Thus line-12 prints "`A 2 digit number`"
- If `num = 5`: (1 digit) `5/1000` evaluates to 0 (integer division) in line-7, and is taken as ***false*** so the `else` statement in line-9 gets executed. Here `5/100` also equals to 0 and is taken as ***false*** so control passes to the `else` in line-11. Here `5/10` again equals to 0 which is again taken as ***false*** so the control passes to the `else` in line-13. Here the only option left is a single digit number. Hence without any further checking, the `printf()` in line-14 prints "`A 1 digit number`"



Conditions without relational operators

Any non-zero value is in general taken as True logic and only zero is taken as False logic in C

You can have a logical condition without using relational operators

Thus in program-24, the values of the expressions evaluated within the conditions are **sufficient to make any decision** and it is redundant to write for example `(num/100>0)`. In the first case `num/1000` evaluates to 2, which is a **non-zero value** and hence is **treated as true**. The condition being satisfied (i.e. true) the `if` part is executed, the `else` part is neglected and the program will print "`A 4 digit number`". Similarly for the other conditions, each condition evaluates to a non-zero value or zero. In case of **non-zero** value it is taken as **True** and in case of a **zero** value it is taken as **False**.



Logical operators

10.4 The Logical Operators (AND, OR, NOT)

When **more than one condition needs to be tested** (as in program-22) to arrive at a decision, then nested `if-else` statements can be used as we have seen earlier. However instead of using multiple levels of `if-else`, which can be at times much confusing and lengthy and giving rise to errors, one can take help of the **Logical Operators**. C provides us with **three logical operators**, which do the same logical functions in C as with Boolean functions. Of these, the **AND and OR** operators are used **to join two or more conditions**. These are the following:

Priority ↑

- `!` : The logical **NOT** operator, it simply **inverts the logic** i.e. !True = False and !False = True
- `&&` : The logical **AND** operator, here the logic is True **only if all the joined conditions** are True
- `||` : The logical **OR** operator, here the logic is True **if any one of the joined conditions** is True

Logical NOT (!) inverts a given logic

OR (||) logic is True if any one of the conditions is True

AND (&&) logic is True if all the individual conditions are True

Logical NOT has the highest priority

Of these operators, the **unary operator** '`!`' i.e. **NOT** has the **highest priority** followed by AND, then by OR. This order helps one to correctly evaluate an expression when more than one operator is present in an expression. The table on the right shows the result of joining multiple conditions using the AND and OR operators.

Condition-1	Condition-2	Logical OR	Logical AND
False	False	False	False
False	True	True	False
True	False	True	False
True	True	True	True



Let us first see the **use of the logical OR operator ||**. It can be typed by pressing the **Shift key** and the key at the top right of the **Enter** key on the keyboard twice (see figure). We write a program to test **if a number entered is an Armstrong number or not**. An Armstrong number is a **three digit number** where the sum of the cube of the digits of the number is equal to the number. There are 4 numbers that satisfy this condition. These are 153, 370, 371, and 407 [$153 = 1^3+5^3+3^3$, $371 = 3^3+7^3+1^3$, etc.]

First we write the program **without using any logical operator** and using an `if-else` ladder to check with each value separately.

```
1   /*Program-25: Checking multiple options without using logical OR operator*/
2   include<stdio.h>
3   int main()
4   {int num;
5    printf("\nEnter a 3 digit integer value: ");
6    scanf("%d", &num);
7    if(num==153)
8        printf("\nArmstrong number");
9   else  if(num==370)
10       printf("\nArmstrong number");
11  else  if(num==371)
12       printf("\nArmstrong number");
13  else  if(num==407)
14       printf("\nArmstrong number");
15  else
16       printf("\nNot an Armstrong number");
17  return 0;
18  }
```

Output1:

```
Enter a 3 digit integer value: 542
Not an Armstrong number
```

Output2:

```
Enter a 3 digit integer value: 371
Armstrong number
```

The same program can be written in a more **compact manner** using logical OR operator || as shown below:



Use of Logical OR operator

Logical AND and OR can be used to join multiple conditions

```
1  /*Program-25a: Use of logical OR operator*/
2  include<stdio.h>
3  int main()
4  {int num;
5   printf("\nEnter a 3 digit integer value: ");
6   scanf("%d", &num);
7   if(num==153 || num==370 || num==371 || num==407)
8       printf("\nArmstrong number");
9   else
10      printf("\nNot an Armstrong number");
11  return 0;
12 }
```

Note how the logical OR operator (**||**) is used to connect the four conditions in line-7. Being an OR operation, if any one of the conditions is True, then the overall condition is evaluated True. So if the input number in line-6 is equal in value to any of the four numbers 153, 370, 371, or 407, then the condition in line-7 is True and prints "Armstrong number".

REMEMBER: If the output of an *if-else* ladder logic is same for different conditions then it can be replaced by logical *OR* with each condition under *if-else* logic being connected by an *OR* operator

Note that the statement shown below is incorrect in syntax. Each condition should be written separately and fully as shown in line-7 of the program above.

```
if( num==153 || 370 || 371 || 407 )      /*Wrong syntax of logical || operator*/
```

Next let us see the **use of the logical AND operator &&**. We will check if a two digit number entered by the user is such that the number is divisible by 2, the square of the number is divisible by 3, and the cube of the number is divisible by 5. There are three two digit numbers 30, 60, and 90 that satisfy this condition. As before we will first write the program **without the use of logical AND** operators.

```
1  /*Program-26: Checking multiple options together without using logical AND operator*/
2  include<stdio.h>
3  int main()
4  {int num;
5   printf("\nEnter a 2 digit integer value: ");
6   scanf("%d", &num);
7   if(num%2 == 0)                      /*Condition checks if number is divisible by 2*/
8     { if((num*num)%3 == 0)            /*Condition checks if square is divisible by 3*/
9         {if((num*num*num)%5 == 0)     /*Condition checks if cube is divisible by 5*/
10           printf("\nNumber satisfies the required conditions");
11        else
12           printf("\nNumber does NOT satisfy the required conditions");
13        }
14      else
15          printf("\nNumber does NOT satisfy the required conditions");
16    }
17  else
18      printf("\nNumber does NOT satisfy the required conditions");
19  return 0;
20 }
```

Output1:

```
Enter a 2 digit integer value: 15
Number does NOT satisfy the required conditions
```

Output2:

```
Enter a 2 digit integer value: 60
Number satisfies the required conditions
```

The same program is written in a more **compact manner** using the logical AND operator `&&` as shown below



Use of Logical AND operator

```
1  /*Program-26a: Checking multiple options together using logical AND operator*/
2  include<stdio.h>
3  int main()
4  {int num;
5   printf("\nEnter a 2 digit integer value: ");
6   scanf("%d", &num);
7   if( num%2 == 0 && (num*num)%3 == 0 && (num*num*num)%5 == 0)
8         printf("\nNumber satisfies the required conditions");
9   else
10        printf("\nNumber does NOT satisfy the required conditions");
11  return 0;
12 }
```

Note how the logical AND operator (`&&`) is used to connect the three conditions in line-7. Being an AND operation, the overall condition is evaluated True only when all the conditions are individually True. So if the input number in line-6 is equal in value to any of the three numbers 30, 60, or 90, then all the conditions in line-7 become True and prints "`Number satisfies the required conditions`". Also note that the statement that the number does not satisfy the conditions is written **only once** unlike the previous program where it had to be written at three places.

REMEMBER: In case you have consecutive nested *if* statements all of which need to be satisfied for a given situation, it can be replaced by logical AND operators, connecting those conditions.

Also note that you can always write programs by the proper use of `if-else` nested structures and without using the logical operators. Logical operators make the coding compact in size and easy to understand.

The following program tests whether an input number is a **multiple of either 4 or 5** using logical OR.

```
1  /*Program-27: Program to test multiple of 4 or 5 using logical OR*/
2  #include<stdio.h>
3  int main()
4  {int num;
5   printf("\nEnter the number to check: ");
6   scanf("%d", &num);
7   if ((num%4==0)||(num%5==0)
8      printf("\nNumber is either a multiple of 4 or 5");
9   else
10     printf("\nNumber is not a multiple of 4 or 5");
11  return 0;
12 }
```

Output1:

```
Enter the number to check: 16
Number is either a multiple of 4 or 5
```

Output2:

```
Enter the number to check: 30
Number is either a multiple of 4 or 5
```

Output3:

```
Enter the number to check: 20
Number is either a multiple of 4 or 5
```

Output4:

```
Enter the number to check: 63
Number is not a multiple of 4 or 5
```

Let us analyse the above program for the four different inputs as shown above. **Remember that an OR operation is True if any one of the conditions involved is True**. Consider line-7 i.e. the condition of the `if` statement to understand the logic for each of the values.

- **Input**=16: `((16%4==0)||(16%5==0))` ⇒ `((0==0)||(1==0))` ⇒ `((True)or(False))` ⇒ `True` ⇒ `1`
- **Input**=30: `((30%4==0)||(30%5==0))` ⇒ `((2==0)||(0==0))` ⇒ `((False)or(True))` ⇒ `True` ⇒ `1`
- **Input**=20: `((20%4==0)||(20%5==0))` ⇒ `((0==0)||(0==0))` ⇒ `((True)or(True))` ⇒ `True` ⇒ `1`
- **Input**=63: `((63%4==0)||(63%5==0))` ⇒ `((3==0)||(3==0))` ⇒ `((False)or(False))` ⇒ `False = 0`

Therefore we find that in the **first three cases** either one of the conditions is True or both the conditions are True. Accordingly as per OR logic **the overall condition is True** and the program prints "`Number is either a multiple of 4 or 5`". However in the **last case** both the conditions joined by the OR operation are False. Hence the **overall condition is False** and the `else` statement is executed with the program printing "`Number is not a multiple of 4 or 5`".

The following program shows the use of the NOT or complement operator `!`. As discussed before, it is used to negate or complement a given logic i.e. makes a True logic False and a False logic True. The program inputs two numbers and **checks if the first number is a multiple of the second number**.



Use of Logical NOT operator

```
1  /*Program-28: Program to show use of logical NOT operator*/
2  #include<stdio.h>
3  int main()
4  {int num1, num2;
5   printf("\nEnter the larger number: ");
6   scanf("%d", &num1);
7   printf("\nEnter the smaller number: ");
8   scanf("%d", &num2);
9   if( !( num1 % num2 ) )
10    printf("\n%d is a multiple of %d", num1, num2);
11  else
12    printf("\n%d is not a multiple of %d", num1, num2);
13  return 0;
14 }
```

Output1:

```
Enter the larger number: 16
Enter the smaller number: 4
16 is a multiple of 4
```

Output2:

```
Enter the larger number: 27
Enter the smaller number: 5
27 is not a multiple of 5
```

In the above program if `num1` is a multiple of `num2` then `num1%num2` (remainder of dividing `num1` by `num2`) will be 0. This condition is tested in the `if` statement of line-9. But if the relation was used alone, then in case `num1` is a multiple of `num2`, then the remainder is zero and the condition would have resulted in:

`if(num1%num2)` ⇒ `if(0)` ⇒ `False` ⇒ i.e. it would not execute the statement in line-10.

A way out would be to **interchange** the `if-else` print statements or use the equality condition `if(num1%num2==0)`. Instead we have used the **NOT** operator i.e. `!`, to reverse the logical value, so that when the relation is satisfied i.e. the remainder is 0, we get:

`if(!(num1%num2))` ⇒ `if(!(0))` ⇒ `if(!(False))` ⇒ `if(True)`

sting a range of values

- **To test a Range of Values using logical operators:**

You can use logical operators to check whether a value is within a given range of values or not. The result of an examination is to be printed based on the marks received by the student. The program is to print "**Excellent**" if the marks are equal or above 80%, "**Good**" if marks are from 60% to less than 80%, "**Fair**" if marks are from 40% to less than 60% and "**Poor**" if the marks are below 40%The following program is first written without using logical operators.

```
1  /*Program-29: Program to print remarks based on marks using if-else ladder*/
2  #include<stdio.h>
3  int main()
4  {int percent;
5   printf("\nEnter the percentage of marks in the subject: ");
6   scanf("%d", &percent);
7   if ( percent>=80 )          /*Condition True only if percent is >=80*/
8       printf("\nExcellent"); /*Prints when percent >=80*/
9   else if( percent>=60 )      /*Else percent <80. But condition is True if percent >=60*/
10      printf("\nGood");       /*Prints when percent >=60 but <80*/
11  else if( percent>=40 )      /*Else percent <60. But condition is True if percent >=40*/
12      printf("\nFair");       /*Prints when percent >=40 but <60*/
13  else                        /*Else percent is <40*/
14      printf("\nPoor");       /*Prints when percent <40*/
15  return 0;
16 }
```

al AND tor can be to check a of values

Now we write the same program using **logical AND operators**:

```
1  /*Program-29a: Program to print remarks based on marks using AND operator*/
2  #include<stdio.h>
3  int main()
4  {int percent;
5   printf("\nEnter the percentage of marks in the subject: ");
6   scanf("%d", &percent);
7   if( percent>=80 )                    /*Condition True only if percent is >=80*/
8       printf("\nExcellent");
9   if( (percent>=60) && (percent<80) )  /*Condition True only if percent is >=60 and <80*/
10      printf("\nGood");
11  if( (percent>=40) && (percent<60) )  /*Condition True only if percent is >=40 and <60*/
12      printf("\nFair");
13  if( percent<40 )                     /*Condition True only if percent is <40*/
14      printf("\nPoor");
15 return 0;
16 }
```

Output:

```
Enter the percentage of marks in the subject: 55
Fair
```

Let us analyse the output of the above program with an **input percentage of 55**. This number falls within the number range 40 and 60 i.e. the pupil should get a **Fair** grade for this percentage. Let us write down how the conditions evaluate for a marks of 55. As is evident, the first two and last `if` condition fails and the **program executes the third `if` condition only**. The outputs of the four `if` conditions are given below to understand the logic of the AND operators. **Remember that the result of an AND operation is True only if all the conditions involved are True.**

- First condition (line-7): `(55>=80)` ⇒ **False**
- Second condition (line-9): `((55>=60)&&(55<80))` ⇒ **((False)AND(True))** ⇒ **(0 AND 1)** ⇒ 0 ⇒ **False**

- Third condition (line10): `((55>=40)&&(55<60))` ⇒ `((True)AND(True))` ⇒ `(1 AND 1)` ⇒ `1` ⇒ `True`
- Fourth condition (line12): `(55<40)` ⇒ `False`

- **Mixing different logical operators in a condition:**

We can also use a combination of different logical operators in a program. The following program checks if a character input by the user is an alphabet or any other character.

Mixing different logical operators in a condition

The program uses the ASCII values of the alphabets to check the condition. Whatever **character** is typed from the keyboard is **stored in the computer memory as a number**. The number is known as the **ASCII** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange) value of that character.

Note that the upper case alphabets 'A' to 'Z' have **ASCII values from 65 to 90** and the lower case alphabets 'a' to 'z' have **ASCII values from 97 to 122** as displayed in the textbox on the right.

A → 65	a → 97
B → 66	b → 98
C → 67	c → 99
D → 68	d → 100
... ...	... ...
Y → 89	y → 121
Z → 90	z → 122

Each character is internally represented by a number called the ASCII value of the character

Thus if the ASCII value of a character is greater than or equal to 65 and at the same time less than or equal 90 then it is a capital alphabet OR if the ASCII value of a character is greater than or equal to 97 and at the same time less than or equal to 122 then it is a small alphabet (note there is a gap of 6 numbers 91 to 96 between the uppercase and lowercase alphabets).

```
1  /*Program-30: Program to show the use of combined logical operators*/
2  #include<stdio.h>
3  int main()
4  {char letter;
5   printf("\nEnter any character: ");
6   letter=getchar();
7   if( ( (letter>=65) && (letter<=90) ) || ( (letter>=97) && (letter<=122) ) )
8     printf("\nYou have entered an alphabet");
9   else
10    printf("\nYou have entered a symbol or a digit");
11  return 0;
12 }
```

Output1:

```
Enter any character: d
You have entered an alphabet
```

Output2:

```
Enter any character: $
You have entered a symbol or a digit
```

The order in which the conditions are checked in line-7 for the input variable `letter` containing the character 'd' whose ASCII value is 100 is shown below:

```
if(  ( (letter>=65)&&(letter<=90) ) || ( (letter>=97)&&(letter<=122) ) )
```

First the innermost boxes
Then the middle boxes
And finally the outer box is evaluated

```
⇒  if( ( ('d'>=65) && ('d'<=90) ) || ( ('d'>=97) && ('d'<=122) ) )
⇒  if( ( (100>=65) && (100<=90) ) || ( (100>=97) && (100<=122) ) )
⇒  if( ( (True) && (False) ) || ( (True) && (True) ) )
⇒  if( (False) || (True) )
⇒  if( True )
⇒  if(1)
⇒  True        i.e. the program prints "You have entered an alphabet"
```

Note how a **character is automatically converted to its ASCII value** before it is compared. Moreover to express a value as a character it is **enclosed within a pair of single quotes** ' ' as 'd' etc. The braces have been given to **maintain proper priority** (remember AND has higher priority over OR).

Similarly, if someone enters '%' the conditions evaluate as (with `letter = '%'` = ASCII 37):

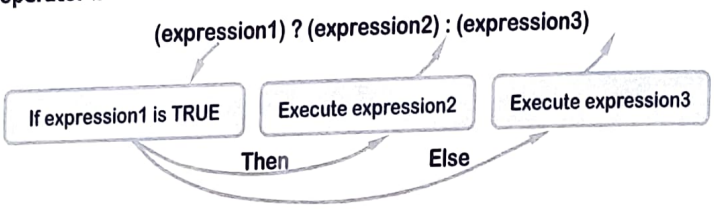
```
    if( ( (letter>=65) && (letter<=90) ) || ( (letter>=97) && (letter<=122) ) )
⇒   if( ( ('%'>=65) && ('%'<=90) ) || ( ('%'>=97) && ('%'<=122) ) )
⇒   if( ( (37>=65) && (37<=90) ) || ( (37>=97) && (37<=122) ) )
⇒   if( ( (False) && (True) ) || ( (False) && (True) ) )
⇒   if( (False) || (False) )
⇒   if(False)
⇒   if(0)
⇒   False        i.e. the program prints "You have entered a symbol or a digit"
```

REMEMBER: In an expression having all types of operators, the Arithmetic operators are evaluated first, followed by Relational operators, followed by Logi cal operators. Thus: **Arithmetic > Relational > Logical.**

10.4 The Conditional Operator (? :)

The working of the conditional operator is similar to that of an `if-else` statement, but with certain limitations. It is a **ternary operator** as it requires **three arguments**. The general format of the **conditional operator** is as follows:

Structure of conditional operator



Conditional operator is also called Ternary Operator as it takes three arguments.

As can be seen from the above diagram, depending upon the logic of the first expression i.e. **expression1**, if it is True then **expression2** will get executed. On the other hand if it is False then **expression3** will get executed. Note that each expression is enclosed within braces.

Though the operation is similar to that of an `if-else` statement, however in the conditional operator **each expression can hold only a single statement** whereas each of the `if` and `else` blocks can hold multiple statements. Though multiple statements are not possible while using the conditional operator, but **nesting of one conditional operator within another is possible**.

Use of conditional operator

The following example illustrates the use of the conditional operator to find the greater of two numbers.

```
1   /*Program-31: To find the greater of two numbers using conditional operator*/
2   #include<stdio.h>
3   int main()
4   {int a, b, max;
5    printf("\nEnter number1: ");
6    scanf("%d", &a);
7    printf("\nEnter number2: ");
8    scanf("%d", &b);
9    max = (a>=b) ? (a) : (b);
10   printf("\nThe greater of the two numbers is %d", max);
11   return 0;
12  }
```

Output1:

```
Enter number1: 16
Enter number2: 5
The greater of the two numbers is 16
```

Output2:

```
Enter number1: 6
Enter number2: 15
The greater of the two numbers is 15
```

Output3:

```
Enter number1: 7
Enter number2: 7
The greater of the two numbers is 7
```

In the above example, the two numbers **a** and **b** are input in line-6 and 8 respectively. The greater amongst these two numbers is then stored in the variable **max**. This greater is calculated using the conditional operator. Let us now understand how the conditional operator in line-9 is executed for the three different inputs. Here **exression1** is the condition **a>=b**, **expression2** is equal to **a** and **expression3** is equal to **b**.

- Number1=16, Number2=5: `(a>=b)?(a):(b)` ⇒ `(16>=5)?(16):(5)` ⇒ `(True)?(16):(5)` ⇒ `16`
- Number1=6, Number2=15: `(a>=b)?(a):(b)` ⇒ `(6>=15)?(6):(15)` ⇒ `(False)?(6):(15)` ⇒ `15`
- Number1=7, Number2=7: `(a>=b)?(a):(b)` ⇒ `(7>=7)?(7):(7)` ⇒ `(True)?(7):(7)` ⇒ `7`

If **a** is greater than **b**, then **expression1** is **satisfied** and hence **expression2** gets executed with the program assigning **16** to **max**, for the input set1. If **b** is greater than **a**, then the **expression1** is **not satisfied** and hence **expression3** gets executed with the program assigning **15** to **max**, as with the input set2. Finally for the input set3, the condition in **expression1** is satisfied and the program executes **expression2** with the value **7** being assigned to **max**.

To do the same job using an **if-else** block the program code would have looked like:

```
1  if(a>=b)
2    max = a;
3  else
4    max = b;
```

The next program shows the use of the conditional operator in another way for doing the same thing as in the previous problem. It displays the greater of two numbers.

```
1  /*Program-32: To display the greater of two numbers using conditional operator*/
2  #include<stdio.h>
3  int main()
4  {int a, b;
5   printf("\nEnter number1: ");
6   scanf("%d", &a);
7   printf("\nEnter number2: ");
8   scanf("%d", &b);
9   (a>=b)?( printf("\n%d is greater", a) ):( printf("\n%d is greater", b) );
10  return 0;
11 }
```

Output1:

```
Enter number1: 16
Enter number2: 5
16 is greater
```

Output2:

```
Enter number1: 6
Enter number2: 15
15 is greater
```

Here the conditional operator executes by itself based on the condition. The **result** of the conditional operator is **not assigned to any variable** like **max**. But depending upon the condition, either **expression2** is executed when the program prints "**16 is greater**", or **expression3** gets executed when the program prints "**15 is greater**" as shown in the first output. Note that we can use any valid C statement as **expression2** or **expression3**. In this case we have used the **printf()** function as the arguments. But always remember that **you can use only a single statement**.

Let us now **redo Program-17** using the conditional operator i.e. to input the total purchase value of a customer and **calculates the net amount payable** after offering different discount values depending upon the amount.

```
1   /*Program-33: Program to calculate different discounts using conditional operator*/
2   #include<stdio.h>
3   int main()
4   {float amount, total;
5    printf("\nEnter the total purchase amount (Rs.): ");
6    scanf("%f", &amount);
7    (amount<3000.0) ? (total=0.9*amount) : (total=0.8*amount);
8    printf("\nThe net amount payable = Rs. %0.2f", total);
9    return 0;
10  }
```

In the above program, depending upon the purchase value, the condition in line-7 is either True or False. In case the condition i.e. `amount<3000.0` is True then the `total` is calculated as `0.9*amount`, else if it is False then the `total` is calculated as `0.8*amount`. As long as it is a single statement, **calculations can also be done as part of the expressions in the conditional operator**.

Conditional operator can also be placed on the left of the assignment operator

Also note that in line-7, the value in `amount` is compared with the value `3000.0` and not with the integer value `3000`. This is because, the variable `amount` that is compared has been declared as a floating point number in line-4. Hence to get the correct result it should be compared with a floating point value only. Hence the intege r value `3000` is converted to a floating point value by putting the decimal point and the `0`. Only a decimal point after `3000` as `3000.` would have also done the job.

Conditional operator on the left side of assignment operator

The **conditional operator** can also be **used in the left side of the assignment operator**. The following example shows how it can be done.

```
1   /*Program-34: Conditional operator on the left side of assignment operator*/
2   #include<stdio.h>
3   int main()
4   {double amount, total; int a=0, b=0;
5    printf("\nEnter purchase amount: ");
6    scanf("%lf", &amount);
7    (amount>=5000.0)?(a):(b) = 1;
8    total = amount - a*0.4*amount - b*0.2*amount;
9    printf("\nTotal amount payable is %0.2lf", total);
10   return 0;
11  }
```

Output1:

```
Enter purchase amount: 6540
Total amount payable is 3924.00
```

Output2:

```
Enter purchase amount: 2460
Total amount payable is 1968.00
```

The above program calculates the discounted amount depending upon the total purchase amount by a customer. Two variables `a` and `b` are initialised to zero in line-4. The purchase amount is checked using the conditional operator of line-7. In case the `amount` is more than or equal to `5000`, the conditional operator evaluates to `a`. The resultant statement is then similar to:

```
a = 1; /*The constant value 1 then gets assigned to variable a*/
```

Otherwise the conditional operator evaluates to `b`. The resultant statement in that case is similar to:

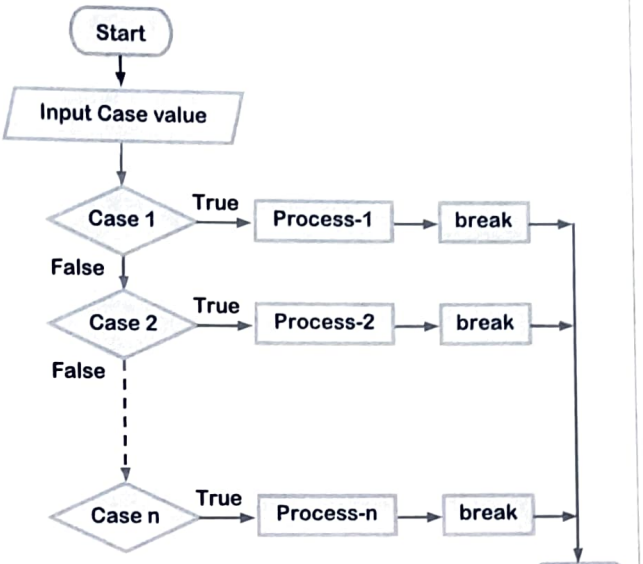
```
b = 1; /*The constant value 1 then gets assigned to variable b*/
```

Depending upon the value of `a` and `b` (i.e. 0 or 1), either a 40% discount value or a 20% discount value gets subtracted from `amount` in line-8 and the result stored in variable `total` and printed in line-9.

10.5 The switch–case–default statement

To select from multiple branching, apart from using nested `if-else` statements, C provides us with another special type of construct called the `switch-case-default` construct. The particular construct successively **tests the value of an expression or variable against a list of integers or character constants**. When a match is found, the statements associated with that constant are executed.

However `switch-case` can **only check for an equality** and a `case` can have **only integer or character constants** (*floats are not allowed*), whereas `if-else` can check for any logical or relational expression and can include a `float` value. The flowchart is given on the right and the syntax for the `switch-case` construct is given below:




```
switch(integer_or_character_variable)
 {case constant1:   statements;
  case constant2:   statements;
  case constantn:   statements;
  default:          statements;
 }
```

NO semicolon after switch

Each case constant followed by a colon :

switch-case-default structure

switch-case control structure is normally used for a menu driven program

switch-case control structure can replace an if-else ladder logic

A switch-case control structure can have an int or char type argument only.

When a `switch` statement is run, the value contained in the integer or character variable after the `switch` keyword is matched with each of the constant values following the `case` keywords. As soon as the **value matches** with a `case` constant, the **statements for that** `case` on the right of the colon (:) get executed.

However, **once a matching is found** with any one of the case constants, apart from the statements associated with that particular case constant, **all the other statements associated with the subsequent cases along with the** `default` **statement also get executed**.

The `break` **statement can be used** after the statements associated with a given `case` constant **to** `break` **out of** `switch-case` from that point. It helps to execute only the statements associated with a given `case` constant to execute, without executing any remaining statements after that. We will discuss the use of the `break` statement in detail in the next chapter dealing with iteration or loops.

If **no match** is found, the statements following the `default` (which is optional) statement get executed.

The following features of the switch-case-default structure should be noted:

1. One can use **only integer or character** values for the `case` constants. Floating point variables are not allowed as a `case` constant
2. One can have only equality of cases and we cannot have any other inequality operation (like>, < etc.)
3. Each `case` constant is followed by a colon (:)
4. There can be multiple statements following a `case` constant and these statements need not be enclosed within a pair of curly brackets { }
5. The `default` statement is optional and can be placed anywhere within the `switch` block
6. After entering the `switch` block each `case` constant is matched with the `switch` variable. If a match is not found with a `case` constant, then the control skips to the next `case` constant
7. As soon as the `switch` value matches with a `case` constant, the statements following that `case`, and all statements associated with the subsequent `cases` along with the `default` statement get executed

Features of switch-case

The default statement in a switch-case control structure is optional.

8. To execute only the statements following a particular **case**, a **break** statement is included after the statements following the **case** constant. Once a **case** is satisfied, the control simply falls through the subsequent statements till it reaches a **break** statement
9. A switch-case-default structure is generally used **to write a menu driven program**, where each option of the menu corresponds to a **case** constant

The following program illustrates the use of **switch-case-default** by creating a simple calculator.



Use of switch-case

```
/*Program-35: Simple Calculator using switch-case*/
#include<stdio.h>
int main()
{ int option, check=1; float a, b, result;
  printf("\nEnter first number: ");
  scanf("%f", &a);
  printf("\nEnter second number: ");
  scanf("%f", &b);
  printf("\nPress 1 to add, 2 to subtract, 3 to multiply, 4 to divide, 5 to exit: ");
  scanf("%d", &option);
  switch(option)
     {case 1 :    result = a+b;
                  break;
      case 2 :    result = a-b;
                  break;
      case 3 :    result = a*b;
                  break;
      case 4 :    if(b!=0)
                     result = a/b;
                  else
                    {printf("\nDivision by 0 not allowed!");
                     check=0;
                    }
                  break;
      case 5 :    break;
      default:    printf("\nYou have not entered a valid choice!");
                  check=0;
     }
  if (check == 1) printf("\nThe required result = %f", result);
  return 0;
}
```

A break statement is used to forcibly come out of the switch-case control structure.

If any case value becomes True then by default all the subsequent case statements will get executed.

Output1:

```
Enter first number: 2.5
Enter second number: 0.5
Press 1 to add, 2 to subtract, 3 to multiply, 4 to divide, 5 to exit: 2
The required result = 2.000000
```

Output2:

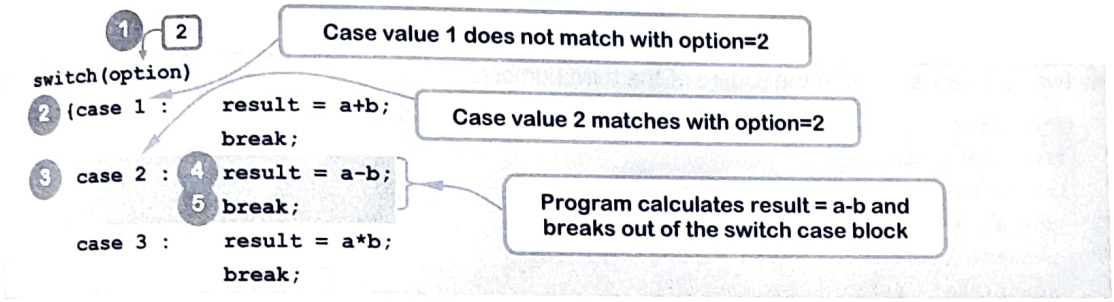
```
Enter first number: 7
Enter second number: 0
Press 1 to add, 2 to subtract, 3 to multiply, 4 to divide, 5 to exit: 4
Division by 0 not allowed!
```

Output3:

```
Enter first number: 7.9
Enter second number: 3.8
Press 1 to add, 2 to subtract, 3 to multiply, 4 to divide, 5 to exit: 7
You have not entered a valid choice!
```

The variables `a` and `b` are input in line-6 and 8. The `option` is input in line-10. Based on the `option` value typed by the user, the `switch-case` block will execute certain part of the code.

For the **Output1**, the `option` chosen is `2`. Hence the operation required is subtraction. The switching occurs based on the value of the variable `option`. After entering the `switch` block in line-11, next in line-12 the `case` value `1` is compared with the `option` value `2`. As these do not match, the control goes straight to the next `case` statement in line-14. In line-14 the `case` value 2 is compared with the `option` value 2. As these values match, the control enters the `case` statements i.e. calculates the result for subtraction and breaks out of the `switch` block by executing the `break` statement in line-15. The diagram below illustrates the process.

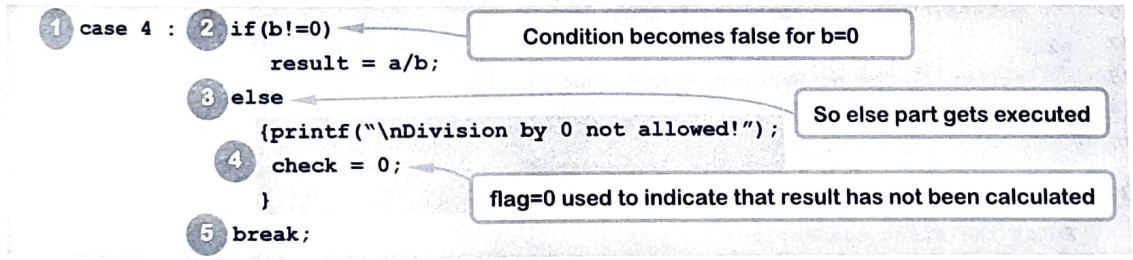


Working of switch-case

In **Output2**, the value of the variable `b` entered is `0`. Hence when `option 4` is given for doing division, the `case` in line-18 gets True. However, since division by `0` is not defined, t he `if` condition in line-18 gets False and the `else` statements get executed. Thus the program executes line-21 and 22. The message that division is not possible is printed in line-21.

Logical AND and OR can be used to join multiple conditions

Let us now understand the use of the variable `check`. It is used to check if a valid result has been obtained or not. In line-4 the variable `check` is initialised to `1`. The value of `check` is changed to `0` in line-22 to indicate that the `result` calculation was not possible. Finally, line-29 checks the value of `check` and as it is equal to `0`, the `printf()` function following it is not executed. The diagram below illustrates the process.



For **Output3**, as the `option` value of 7 does not match any of the `case` values, the `default` statement in line-26 gets executed. As part of this the program prints "`You have not entered a valid choice!`" and changes the value of the variable `check` to `0` to indicate that the `result` could not be calculated.

Difference between If-Else and Switch-Case

If-Else	Switch-Case
The `if` statement tells the compiler that the instruction to follow is a decision and **branch according to the condition of the decision.**	C provides us with a special type of construct called the `switch-case-default` construct to **select from multiple branching.**

The general structure of the `if-else` construct is:
`if (condition is true)`
`{ execute statement set1;}`
`else`
`{ execute statement set2;}`
	The general structure of the `switch-case` construct is:
	`switch(variable)`
	`{case constant_a: statements;`
	`case constant_b: statements;`
	`case constant_n: statements;`
	`default: statements;`
	`}`

`if-else` can check for **any logical or relational expression.**	`switch-case` can **only check for an equality.**

Difference between if-else and switch-case

If-Else	Switch-Case
When an `if` statement is run, if the condition following `if` is true then the statement block after `if` is executed. In case the condition is false, the statement block after the `else` block gets executed.	When a `switch` statement is run, the value contained in the integer or character variable after the switch keyword is matched with each of the constant values following the `case` keywords.
`if-else` can include a `float` value in its condition portion	`switch-case` can have only integer or character constants (floats are not allowed).



Worked out problems

10. 6 Some worked out problems

Input three numbers and **check if the numbers are Pythagorean or not** i.e. if the sum of the squares of any two numbers is equal to the square of the third number.

```
1   /*Program-36: Program to check if 3 numbers are Pythagorean*/
2   #include<stdio.h>
3   int main()
4   {int a, b, c;
5    printf("\nEnter the first number: ");
6    scanf("%d", &a);
7    printf("\nEnter the second number: ");
8    scanf("%d", &b);
9    printf("\nEnter the third number: ");
10   scanf("%d", &c);
11   if ( a*a + b*b == c*c )
12      printf("\nThe numbers are Pythagorean");
13   else if ( b*b + c*c == a*a )
14      printf("\nThe numbers are Pythagorean");
15   else if ( c*c + a*a == b*b )
16      printf("\nThe numbers are Pythagorean");
17   else
18      printf("\nThe numbers are NOT Pythagorean");
19   return 0;
20  }
```

Output1:

```
Enter the first number: 5
Enter the second number: 4
Enter the third number: 3
The numbers are Pythagorean
```

Output2:

```
Enter the first number: 7
Enter the second number: 5
Enter the third number: 9
The numbers are NOT Pythagorean
```

In the above program, the numbers can be entered in any order to check if they are Pythagorean or not. In **Output1**, the condition in line-13 gets satisfied and the program prints "`The numbers are Pythagorean`". However in **Output2**, since the numbers are non-Pythagorean, none of the conditions are true in lines-11, 13 and 15, only the final option in line-18 is left and the program prints "`The numbers are NOT Pythagorean`".

Input the length of three line segments and check if those can be used to form a triangle or not.

```
1   /*Program-37: Program to check if 3 line segments can form a triangle or not*/
2   #include<stdio.h>
3   int main()
```

```
{int a, b, c;
printf("\nEnter the first line segment length: ");
scanf("%d", &a);
printf("\nEnter the second line segment length: ");
scanf("%d", &b);
printf("\nEnter the third line segment length: ");
scanf("%d", &c);
if ( a+b > c )                    /*Checks Condition1*/
   { if ( b+c > a )               /*If Condition1 is True, checks Condition2*/
       { if ( c+a > b )           /*If both Condition1 & Condition2 are True checks Cond.3*/
            printf("\nLine segments form a triangle"); /*When all Condtn. True*/
         else
            printf("\nLine segments CANNOT form triangle"); /*If Condition3 False*/
       }
     else
         printf("\nLine segments CANNOT form triangle");      /*If Condition2 False*/
   }
else
     printf("\Line segments CANNOT form triangle");          /*If Condition1 False*/
return 0;
}
```

Output1:

```
Enter the first line segment length: 5
Enter the second line segment length: 4
Enter the third line segment length: 3
Line segments form a triangle
```

Output2:

```
Enter the first line segment length: 2
Enter the second line segment length: 6
Enter the third line segment length: 3
Line segments CANNOT form triangle
```

Check if a year entered by user is a leap year or not without using logical operators. A millennium year (i.e. a year ending with 00 e.g. 1800, 1900, 2000 etc.) is a leap if it is divisible by 400 and not 4. For other years it should be divisible by 4.

```
/*Program-38: Program to check if year is a leap year or not*/
#include<stdio.h>
int main()
{int year;
 printf("\nEnter an year: ");
 scanf("%d", &year);
 if ( year%400 == 0 )                    /*Checks for millennium years divisible by 400*/
      printf("\nYear is a LEAP year");          /*Prints for millennium leap years*/
 else
   { if ( year%100 == 0 )                /*If millennium year but not divisible by 400*/
        printf("\nYear is NOT a leap year"); /*Prints for millennium non leap years*/
     else if ( year%4 == 0 )             /*Checks if non millennium year divisible by 4*/
        printf("\nYear is a LEAP year");       /*Prints if non-millennium leap year*/
     else
        printf("\nYear is NOT a leap year");   /*Prints if all conditions are False*/
   }
 return 0;
}
```

Output1:

```
Enter an year: 1800
Year is NOT a leap year
```

Output2:

```
Enter an year: 2000
Year is a LEAP year
```

Output3:

```
Enter an year: 2009
Year is NOT a leap year
```

Output4:

```
Enter an year: 2012
Year is a LEAP year
```

Program to find the **location of a Cartesian point** entered by the user, without using logical operators.

```
1  /*Program-39: Program to find the location of a Cartesian point*/
2  #include<stdio.h>
3  int main()
4  {int x, y;
5   printf("\nEnter the co-ordinates of a point separated by blank space: ");
6   scanf("%d%d", &x, &y);
7   if ( x > 0 )                        /*Checks if x is positive*/
8       {if ( y > 0 )                   /*Checks if y is positive*/
9            printf("\nQuadrant-1");    /*If both x, y positive, then quadrant-1*/
10       else if ( y < 0 )              /*Checks if y is negative*/
11           printf("\nQuadrant-4");    /*If x positive, y negative then quadrant-4*/
12       else                           /*If x positive & y neither positive, nor negative*/
13           printf("\nX Axis");        /*i.e. x positive and y is zero then X Axis*/
14      }
15  else
16      { if ( x < 0 )                  /*Checks if x is negative*/
17           {if ( y > 0 )              /*Checks if y is positive*/
18                printf("\nQuadrant-2"); /*If x negative, y positive, then quadrant-2*/
19            else if ( y < 0 )         /*Checks if y is negative*/
20                printf("\nQuadrant-3"); /*If x negative, y negative then quadrant-3*/
21            else                      /*If x negative & y neither positive, nor negative*/
22                printf("\nX Axis");   /*i.e. x negative and y is zero then X Axis*/
23           }
24        else                          /*Only option left for x is x equal to 0*/
25           { if ( y != 0 )            /*If y not equal to 0*/
26                printf("\nY axis");   /*If x is 0 and y not equal to 0, then Y Axis*/
27             else                     /*Else if y equal to 0*/
28                printf("\nOrigin");   /*Both x equal to 0 and y equal to 0, hence Origin*/
29           }
30      }
31  return 0;
32 }
```

Output1:

```
Enter the co-ordinates of a point separated by blank space: -2  5
Quadrant-2
```

Note that under the `if` block of line-7, an **`if-else`** ladder is used to check the possibilities when **x** is greater than zero. Within the `if` block of line-16 another **`if-else`** ladder is used to check the possibilities when **x** is less than zero. Finally the **`else`** block of line-24 is used to check the remaining possibilities when **x** is equal to zero.

The following program uses **switch-case-default** statement to print different area values.

```
1  /*Program-40: Calculating different area values using switch-case*/
2  #include<stdio.h>
3  int main()
4  { char option; float s, area;
5    printf("\nPress 'a' for Circle, 'b' for Square, 'e' to Exit: ");
6    scanf("%c", &option);
7    switch(option)
8       {case 'A' :  /*Remember to put single quotes for character constants as 'A' etc*/
9        case 'a' :  puts("\nEnter radius of circle: "); scanf("%f", &s);
10                   area = 3.14159*s*s;
11                   printf("\nThe required area of circle = %0.2f", area);
12                   break;
13       case 'B' :
14       case 'b' :  puts("\nEnter side of square: "); scanf("%f", &s);
15                   area = s*s;
16                   printf("\nThe required area of square = %0.2f", area);
17                   break;
18       case 'E' :
19       case 'e' :  break;
20       default  :  puts("\nYou have not entered a valid choice!");
21      }
22    return 0;
23 }
```

Output1:

```
Press 'a' for Circle, 'b' for Square, 'e' to Exit: B
Enter side of square: 2.5
The required area of square = 6.25
```

Output2:

```
Press 'a' for Circle, 'b' for Square, 'e' to Exit: a
Enter radius of circle: 6.4
The required area of circle = 128.69
```

In **Output1**, when the user enters 'B' as the choice, the **case** value in line-13 becomes True and the control goes to the right side of the colon (:) in line-13. As there are no statements to execute in that line and there is no break statement either, the control automatically comes to the next line on the right side of the colon for case value 'b'. There it executes the **puts()** statement and the subsequent lines to calculate and print the area of the square. The **break** statement of line-17 then takes the control out of the **switch-case** structure. Thus if the user enters either 'B' or 'b' in both cases finally the area of the square is calculated.

C to it that .....

The following examples demonstrate the areas where mistakes are common. The <u>whole program is not written always</u> and only the relevant parts are shown.

This program portion checks a number to be even or odd, and if the number is even then checks, if it is a multiple of 3 i.e. it checks for all even multiples of 3.

```
1  {int num;
2   printf("\nEnter a number: ");
3   scanf("%d", &num);
4   if(num%2==0)
5      if(num%3==0)
6         printf("\nNumber is even and a multiple of 3");
7   else
8      printf("\nNumber is simply odd");
9   return 0;
10 }
```

Output1:

```
Enter a number: 12
Number is even and a multiple of 3
```

Output2:

```
Enter a number: 25
```

At a first glance it may seem that if the entered number is an even multiple of 3 like 12, then the program will print "**Number is even and a multiple of 3**" and if not it will print "**Number is simply odd**".

In reality **if an odd number is entered the program will print nothing**. The bug lies in the use of the **else** statement in line-7. From the program indentation it may seem that the **else** goes with the first **if** in line-4. In reality it goes with the second **if** in line-5. The general rule is that **else goes with the nearest if**. To make the **else** statement execute as desired, **we will have to place proper braces** as shown.

```
1  {int num;
2   printf("\nEnter a number: ");
3   scanf("%d", &num);
4   if(num%2==0)
5     {if(num%3==0)
6        printf("\nNumber is even and a multiple of 3");
7     }
8   else
9     printf("\nNumber is simply odd");
9   return 0;
10 }
```

Rectified Output1:

```
Enter a number: 12
Number is even and a multiple of 3
```

Rectified Output2:

```
Enter a number: 25
Number is simply odd
```

The next program inputs a floating-point variable and compares it with a **float literal**.

```
1  int main()
2  {float guess;
3   printf("\nGuess the value of pi (up to 2 digits after decimal): ");
4   scanf("%f", &guess);
5   if(guess > 3.14)
6      printf("\nYour guessed value is greater than PI");
7   if(guess == 3.14)
8      printf("\nYour guessed value is equal to PI");
9   if(guess < 3.14)
10     printf("\nYour guessed value is lesser than PI");
11  return 0;
12 }
```

Output1:

```
Guess the value of pi (up to 2 digits after decimal): 3.24
Your guessed value is greater than PI
```

Output2:

```
Guess the value of pi (up to 2 digits after decimal): 3.14
Your guessed value is greater than PI
```

Output3:

```
Guess the value of pi (up to 2 digits after decimal): 3.04
Your guessed value is lesser than PI
```

From the logic of the above program it is evident that if someone guesses PI as 3.14 then the program should print "**Your guessed value is equal to PI**". But contrary to common sense, to your surprise the program will print, "**Your guessed value is greater than PI**". The reason is that a **floating-point literal** value is not stored exactly as it is shown on the screen but **due to precision considerations it is stored as something less than** `3.14` (say 3.135555). Hence the comparison makes `guess>3.14` as True and prints the statement accordingly.

To overcome such a problem, **declare the variable** `guess` **as a** `double` and use the `%lf` format specifier in `scanf()` function to input the number. This is because internally the number `3.14` is stored as a `double`. Another method is to **forcibly make the literal** `3.14` **a** `float`, as shown below:

```
1  int main()
2  {float guess;
3   printf("\nGuess the value of pi (up to 2 digits after decimal): ");
4   scanf("%f", &guess);
5   if(guess > (float)3.14)
6     printf("\nYour guessed value is greater than PI");
7   if(guess == (float)3.14)
8     printf("\nYour guessed value is equal to PI");
9   if(guess < (float)3.14)
10    printf("\nYour guessed value is lesser than PI");
11  return 0;
12 }
```

Rectified Output2:

```
Guess the value of pi (up to 2 digits after decimal): 3.14
Your guessed value is equal to PI
```

This is known as **type casting** whereby the number `3.14` is **converted temporarily** to a specified data type by the syntax `(float)3.14` in line-5, 7 and 9. Thus before making the comparison `3.14` will be converted from a `double` to a `float` and hence will yield the correct result. The other solution is to make the variable a double type and keep the literal values 3,14 as it is without using typecasting.

```
2  {double guess;
3   printf("\nGuess the value of pi (up to 2 digits after decimal): ");
4   scanf("%lf", &guess);
```

Thus be careful when comparing floating point variables within an `if` statement.

The next program piece checks if a point lies on the origin.

```
1  int main()
2  {int x, y;
3   printf("\nEnter x coordinate of point: "); scanf("%d", &x);
4   printf("\nEnter y coordinate of point: "); scanf("%d", &y);
5   if( x=0 && y=0 )
6      printf("\nPoint lies on the origin.");
7   else
8      printf("\nPoint does not lie on the origin.");
9   return 0;
10 }
```

Output1:

```
Enter x coordinate of point:  0
Enter y coordinate of point:  0
Point lies on the origin.
```

Output2:

```
Enter x coordinate of point:  5
Enter y coordinate of point:  7
Point lies on the origin.
```

Whatever be the input by the user, the above **program will always print** "`Point lies on the origin`". The reason is that in the `if` condition in line-5, instead of using the **equality operator** (==) to **compare** the two values, the **assignment operator** (=) has been used. This results is assigning the value of `0` to `x` and `y` instead of comparing the same with `0`. Thus `x` and `y` **always become equal to** `0` **and results in a permanent True** condition printing the first line. This is a very common mistake. The line should be rectified as:

```
5   if( x==0 && y==0 )
```

Rectified Program Output1:

```
Enter x coordinate of point:  0
Enter y coordinate of point:  0
Point lies on the origin.
```

Rectified Program Output2:

```
Enter x coordinate of point:  5
Enter y coordinate of point:  7
Point does not lie on the origin.
```

CAUTION: It is a common programming mistake to put '=' instead of '==' to check for equality in a condition and care should be taken to avoid the same.

The following program uses the `switch-case-default` structure to find either the square or the cube of a number input by the user.

```
1   int main()
2   { int option, check=1; float a, result;
3     printf("\nEnter number: ");
4     scanf("%f", &a);
5     printf("\nPress 1 to find square, 2 to find cube: ");
6     scanf("%d", &option);
7     switch(option)
8        { case1  :   result = a*a;
9                     break;
10         case2  :   result = a*a*a;
11                    break;
12         default:   printf("\nYou have not entered a valid choice!");
13                    check=0;
14       }
15    if (check == 1) printf("\nThe required result = %f", result);
16    return 0;
17  }
```

Output:

```
Enter number: 52.39
Press 1 to find square, 2 to find cube: 2
You have not entered a valid choice!
```

Even though a valid input choice has been given, yet the program prints for an invalid choice. The fault lies in the use of the `case` keywords. By mistake the choice values have been appended at the end of the `case` keyword as `case1`, `case2`, instead of writing them separately. The rectified code is given below:

```
    switch(option)
    { case 1  : result = a*a;
                break;
      case 2  : result = a*a*a;
                break;
      default:  printf("\nYou have not entered a valid choice!");
                check=0;

    }
```

Rectified Output:

```
Enter number: 52.39
Press 1 to find square, 2 to find cube: 2
The required result = 143795.468750
```

The following program uses the **switch-case-default** structure to wish "Good morning" or "Good Night".

```
int main()
{ char option;
  printf("\nPress 'm' for Morning, 'n' for Night: ");
  scanf("%c", &option);
  switch(option)
    { case m :  puts("Good Morning");
                break;
      case n :  puts("Good Night");
                break;
      default:  printf("\nYou have not entered a valid choice!");
    }
  return 0;
}
```

Output:

```
'a' undeclared (first use in this function)
'b' undeclared (first use in this function)
```

The program will not compile and will give the error message as shown above. The reason is that the case constants have not been used properly. As the case constants are character constants, these should be enclosed within single quotes. The corrected code will have:

```
  switch(option)
  { case 'm' : puts("Good Morning");
               break;
    case 'n' : puts("Good Night");
               break;
    default  : printf("\nYou have not entered a valid choice!");
  }
```

REMEMBER that a **logical expression is evaluated from left to right**. In case sufficient logical truth has been obtained to evaluate an expression, then further evaluation of the expression is stopped and the evaluation of the remaining expressions is not carried out. This is common for conditions where there are logical OR operators. If the first condition is satisfied then the remaining conditions connected by the OR operators is not checked as (True) OR (Anything) is True. However, in case you have AND operations, then all the AND connected conditions are checked to see if the overall condition is True or not.

The next program piece illustrates the above point.

```
int main()
{int a=0, b=0, c=0, x=2, y=4, z=8;
puts("Enter three values:");
```

```
4   scanf("%d%d%d", &x, &y, &z);
5   if( (a=x) && (b=y) || (c=z) )
6       printf("\na=%d, b=%d, c=%d", a, b, c);
7   else
8       printf("\na=%d, b=%d, c=%d", a, b, c);
9   return 0;
10  }
```

Output:

```
Enter three values: 2  5  8
a=2, b=5, c=0
```

The condition of the `if` statement in line-5 consists of assignment operations (note, these are not equality operations). Remember that **C considers any non-zero value as True** and only a zero value as False. The condition evaluates by taking into consideration this property without the use of relational operators.

Ideally to check the overall condition all the individual conditions need to be checked. However, as `&&` operator has a higher priority than `||` operator, the AND condition `(a=x)&&(b=y)` is evaluated to (Non-Zero)&&(Non-Zero) i.e. True&&True which is True. As the next condition is an OR operation, and the left hand argument of the OR operation has been evaluated as True, C will not evaluate or check the right side condition of the OR operation. So the variable `c` remains at zero (0) without any change, though the other variables `a` and `b` both get changed. This is why we get the above output for the values entered.



The Fact File

- **In a program depending upon some condition(s) whenever there are more than one possible outputs for a particular set of input(s), the situation is called program branching**
- To test or verify conditions C provides us with some special operators called relational operators that allow us to compare two values and find out whether one value is Equal (==), Not Equal (!=), Greater Than (>), Less Than (<) etc. to another value.
- **The outcome of a relational operation is either True or False**
- C uses the if-else statement to check a condition and execute statements accordingly.
- **The else statement is dropped in case there are no statements to execute if the condition is false**
- An if statement can exist without an else, but an else statement cannot exist without an if.
- **If there is only one statement to execute in case the condition is true, there is no need to put the curly braces and the statement can be put beside the condition, or in the next line.**
- A True condition has a numeric value of 1 and a False condition has a numeric value of 0
- **All positive and negative values are taken as True and only a 0 is taken as False within a condition.**
- Placing one if-else block within another is known as nesting of if-else blocks.
- **Multiple nested if-else structures can take the shape of a ladder and is hence known as an if-else ladder.**
- Nested if-else statements can be used when more than one condition needs to be tested to arrive at a decision
- **C provides us with three logical operators – NOT, AND, OR**
- The unary logical operator '!' i.e. NOT has the highest priority followed by AND, followed by OR.
- **A NOT logic simply inverts a given logic.**
- An AND logic (&&) is True when all the individual logics are True.
- **An OR logic (||) is True if any one of the conditions is True.**
- Instead of using multiple levels of if-else, which can be at times much confusing and lengthy, one can take help of the Logical Operators to connect several conditions under a single condition statement
- **Whatever character is typed from the keyboard is stored in the memory of the computer as a number. This number is known as the ASCII (American Standard Code for Information Interchange) value of that character.**
- The characters A to Z have ASCII values from 65 to 90 and the characters a to z have ASCII values from 97 to 122.
- **A character is automatically converted to its ASCII value before it is compared.**
- The conditional operator (?:) is also called a ternary operator and works similar to an if-else statement
- **Unlike an if-else statement, each expression in the conditional operator can hold only a single statement whereas each of the if and else blocks can hold multiple statements.**

- To select from multiple branching C provides us with a special type of construct called the switch-case-default construct. It successively tests the value of an expression or variable against a list of integers or character constants. When a match is found, the statements associated with that constant are executed.
- **In a switch-case-default structure, one can use only integer or character values for the case constants. Floating point variables are not allowed as a case constant.**
- One can have only equality of cases and we cannot have any other in-equality (like>, < etc.) in a switch statement.
- **The default statement is optional and can be placed anywhere within the switch block.**
- A switch-case-default structure is generally used to write a menu driven program, where each option of the menu corresponds to a case constant.

Review Questions





Q1. Multiple Choice Questions. Select from any one of the four options. **1 each**

i) Program branching helps us to:
a. Run a code multiple number of times
b. Execute different codes based on conditions
c. Execute all the lines of code in a program
d. Use different types of data types in a program

ii) An if statement in C:
a. May have an else statement
b. Always have an else statement
c. Must have an else statement
d. Can have multiple else statements

iii) How many relational operators are there in C?
a. 3 b. 4 c. 6 d. 8

iv) How do you represent the not equal to operator in C?
a. not = b. <> c. != d. =!

v) How do you represent the less than or equal to operator in C?
a. L= b. <= c. =< d. ≤

vi) Multiple nested *if-else* statements can be sometimes replaced by an _____.
a. if-else slope b. if-else step c. if-else stair d. if-else ladder

vii) Which of the following is not a relational operator in C?
a. >= b. == c. ! d. <

viii) Which of the following operations does not change the value of the variable x?
a. x=9 b. x=x-1 c. (5-2)?(x=6):(x=2) d. x==9

ix) An *if-else* ladder can sometimes be replaced by logical _____ operators:
a. AND b. NAND c. OR d. NOT

x) Nested *if* statements can sometimes be replaced by logical _____ operators:
a. NOT b. OR c. NAND d. AND

xi) Which of the following is not a logical operator in C?
a. & b. ! c. && d. ||

xii) A conditional operator:
a. has 1 argument b. has 2 arguments c. has 3 arguments d. has 4 arguments

xiii) Which of the following data types cannot be a case constant in a switch-case statement?
a. int b. float c. long int d. char

xiv) Which statement in C can be used to come out of a case statement in a switch-case operation?
a. end b. exit c. continue d. break

xv) What will be the output of the C program piece given below?

```
int x=2;
if(x-2==0) if(x/2==1) if(x*2==4) if(x+x==4) if(x=0) puts("Humpty"); else puts("Dumpty");
```

a. Humpty b. Dumpty c. Show error d. None of these

xvi) What will be the output of the following piece of code in C?

```
int x=2, y = 3, z=6, w;
w = (x)||(x-y)&&(x*y==z);
printf("%d", w);
```

a. 1 b. 2 c. 3 d. 6

xvii) What will be the output of the following piece of code in C?

```
int x=5, y = 8, z=12, w;
w = !(x)||(!z==!5)&&(z!=!x)+1;
printf("%d", w);
```

a. 12 b. 1 c. 5 d. 2

xviii) What is the output of the following code in C if the user inputs x as 6?

```
int x, y = 3, m; scanf("%d", &x);
m=(x-y)?(x+y):(x-y);
printf("%d", m);
```

a. 3 b. 9 c. 6 d. 2

xix) What is the output of the following code in C if the user inputs m as 2?

```
int x=12, y = 3, m; scanf("%d", &m);
 (x-4*y)?(x):(y)=m;
printf("%d, %d", x, y);
```

a. 3, 12 b. 12, 3 c. 2, 3 d. 12, 2

xx) What will be the output of the following code in C, if the user inputs 2 for x?

```
int x, y = 3; scanf("%d", &x);
if(x<0)
  printf("\n x=%d", x);
  y=y*(-1);
if(x>0) y=y+6;
  printf("\n%d", y);
```

a. 3 b. 5 c. 9 d. 4

xxi) What will be the output of the following code in C?

```
char x='P'; int A=97, Z=122;
if(x>=A && x<=Z) printf("\nUppercase alphabet");
else printf("\nSomething else");
```

a. Something else b. else without if (error) c. Uppercase alphabet d. None of these

xxii) What will be the output of the following code in C?

```
int x=45, y=45; if( x=x/10 || y=y%10 ) printf("x=%d, y=%d", x, y);
```

a. x=45, y=4 b. x=4, y=45 c. x=45, y=45 d. x=4, y=4

xxiii) What will be the output of the following code in C?

```
int x=5; if(x==2); printf("Hello, Good Morning"); if(x==3) printf("Hello, Goodnight");
```

a. Hello, Goodnight b. Nothing c. Hello, Good Morning d. None of these

xxiv) What will be the output of the following code in C?

```
int x=5; if(x=2) printf("Hello, Goodnight"); else printf("Hello");
```

a. Hello b. Hello, Goodnight c. Compilation Error d. None of these

xxv) What will be the output of the following code in C when the user enters the input x as 5.7?

```
float x; scanf("%f", &x); if(x= =5.7) puts("Abrake"); else puts("Dabra");
```

a. Abrake b. Compilation error c. Dabra d. None of these

xxvi) What will be the output of the following code in C?

```
int x=5; if(x=4) printf("Hotch"); else printf("Potch");
```

a. Potch b. Hotch c. Compilation error d. None of these

xxvii) What will be the output of the following code in C?

```
int x=12, y=3;
if (!(x!=12) && !(y!=15) && (y==3 || x<2 && x<y)) printf("%d", 3*x+1);
else printf("%d", y-1);
```

a. 36 b. 37 c. 3 d. 2

xviii) What will be the output of the following code in C?

```
int x=15, y=2;
if (!(x!=15) && !(y!=2) && (x>-2 || y==15 && x>y)) printf("%d", 2*x-1);
else printf("%d", y+1);
```

a. 2 b. 29 c. 30 d. 3

xix) What will be the output of the following code in C?

```
int x=2, y=10;
if (x-y) printf("%d", x);
else if (y-x) printf("%d", y);
else if (5*x-y) printf("%d", y-x);
else printf("%d", x-y);
```

a. -8 b. 8 c. 10 d. 2

xx) What will be the output of the following code in C?

```
int x=2, y=10;
if (x-y) printf("%d", x);
else if (y-x) printf("%d", y);
else if (5*x-y) printf("%d", y-x);
else printf("%d", x-y);
```

a. -8 b. 8 c. 10 d. 2

Short Answer type questions: **1 each**



i) What types of outputs are possible when a condition is evaluated by an *if* statement?
ii) Name any two relational operators in C.
iii) What is the use of the equality operator in C?
iv) State one difference between a logical not and the inequality operator.
v) State the order in which logical, arithmetic, and relational operators are evaluated in an expression having all of these types of operators.
vi) How many relational operators are present in C?
vii) State one difference between the '=' and the '==' operators.
viii) What do you mean by a nested if-else statement?
ix) State one use of logical operators.
x) Write the operator symbols for logical AND and OR operators in C.
xi) Which logical operator has the highest priority amongst logical operators?
xii) Name the ternary operator used to check conditions in C.
xiii) What is the use of the default statement in a switch-case-default statement?
xiv) What is the value of m in the C code m=(6==2*5-4)?(5):(10);
xv) What is the output of the C statement: *if(5+5) if(4/2) if(8%5) if(8-4*2) puts("1"); else puts("2");*

Long Answer type questions: **7 each**



i) Write a program to show the use of an *if* statement without the use of relational operators. Explain the difference between the '=' and '==' operators with the help of proper examples. What do you mean by an *if-else* ladder? 3+2+2

ii) Show an use of the conditional operator in C. What are the uses of logical operators in C? State two differences between an if-else and a switch-case operation in C. 3+2+2

iii) Show a use of the conditional operator on the left side of the assignment operator with the help of a program. What is the utility of using switch-case operation? Explain how you can check using logical operators whether a given value is within a given range of values or not? 3+2+2

Assignment Programs: **4 each**



i) Input a pair of simultaneous equations and determine if they can be solved to give unique values. If they can be solved then find the solution. Otherwise state that the equations cannot be solved.

[Hint: Take the equations as: **a1**x + **b1**y + **c1** = 0 and **a2**x + **b2**y + **c2** = 0

Input the coefficients a1, b1, c1, a2, b2, c2. In case a1.b2 – a2.b1 = 0, the equations cannot be solved. Check this condition and state whether the equations can be solved or not. If solvable, then find the values of the variables x and y. The variables x and y are given by:

$$x = (b1.c2 - b2.c1)/(a1.b2 - a2.b1), \qquad y = (c1.a2 - c2.a1)/(a1.b2 - a2.b1)$$

ii) Write a program to find the maximum value between two numbers entered by the user.

iii) Use the conditional operator to find the maximum out of three numbers input by the user.

iv) Write a program in C to test if three points are collinear or not. Input the three points as (x1, y1), (x2, y2), (x3, y3). [Hint: For three points to be collinear, the area enclosed by the points will be zero.]

v) Enter the co-ordinate of a point (x, y). Write a C program to check with the help of Logical Operators if it lies in the first, second, third, or fourth quadrant, on x or y axis, or on the origin, and display a message accordingly.

vi) Write a program with the help of Logical Operators to check if a year entered by the user is a leap year (a millennium year like 1800, 1900, 2000, 2100 etc. is a leap year if it is divisible by 400).

vii) Write a program using Logical Operators to check if three line segments can form a triangle.

viii) Write a program using Logical Operators to check if three numbers are Pythagorean triplets.

ix) Write a program in C to test if a point (x, y) lies inside, on or outside a circle of radius r with origin at the centre, where x, y, and r are input by the user.

x) Write a program in C to test if a point (x, y) lies inside, on or outside a circle of radius r with origin at (a, b) where x, y, r, a and b are input by the user.

xi) Write a program to check if a number is even or odd with the help of conditional operators.

xii) Write a program to check if a number is even or odd with the help of *switch-case* construct and **without using** any *if-else* construct.

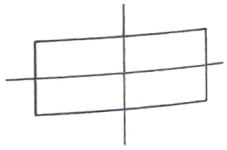
xiii) Input a character. Check if it is in uppercase or in lowercase. If in uppercase, then convert it into lowercase, and if it is in lowercase then convert it into uppercase.

xiv) Input the coefficients a, b and c of a quadratic equation $a.x^2+b.x+c=0$. Check if the equation has real roots. In case it has real roots then calculate the same. [Hint: For real roots check if $b^2-4ac>=0$. If so then find roots using the required relation $x1, x2 = \{-b \pm (b^2 - 4ac)^{0.5}\}/(2.a)$]

To find the square root of b^2-4ac use the sqrt() function. To use it **include the <math.h> header file.** The syntax for using the function is: y = sqrt(d); where $d = b^2-4ac$]

xv) If telephone calls made are less than equal to 300 then charge per call is Rs. 0.80. If calls are between 300 and 601, then rate per call is Rs. 1.20 and if calls are more than 600, then rate is Rs. 1.50 per call. Write a C program to calculate the telephone bill based on the number of calls made by a subscriber.

xvi) A rectangle is centred at the origin and has a length L and a breadth B (entered by the user) as shown in the diagram. Check whether a point (x,y) also entered by the user lies inside, on, or outside the rectangle.



xvii) An air-conditioning system of a storage warehouse is to be turned on if one or more of the following conditions occur:

a. The weight of the stored material is less than 100 tons, the relative humidity is at least 60 percent, and the temperature is above 60 degrees;
b. The weight of the stored material is 100 tons or more and the temperature is above 60 degrees;
c. The weight of the stored material is less than 100 tons and the barometer stands at 30 or over.

W designates weight of 100 tons or more.
H designates relative humidity of at least 60 percent.
T designates temperature above 60 degrees.
P designates barometric pressure of 30 or more.

Write a C program to receive the parameters W, H, T and P and based on the conditions a, b, or c, state whether the air conditioning should be started ON or put OFF.

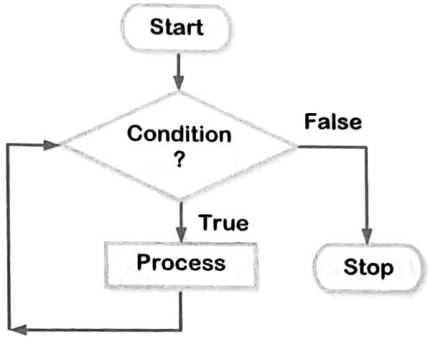
CHAPTER 11
Using Loops in C

- Introduction 11-1
- Concept of a Loop 11-2
- The while Loop 11-6
- The break statement 11-14
- The continue statement 11-15
- The do-while Loop 11-16
- The for Loop 11-18
- Some worked out examples 11-24

11.1 Introduction

A part from conditional branching there may be situations where a particular set of instructions may need to be repeated several times depending upon some conditions until the desired result is got.

As an example take the situation of finding the terms of an A.P. series up to certain number of terms. Each new term is created by adding the common difference to the previous term. This forms a repetitive process. Such a repetitive process is also called **iteration**. The number of times to repeat usually depends upon conditions.



Flowchart for looping

The general logic for such a situation is given by the **flowchart** on the right. As long as the condition is True, the **Process part is repeated**. Each time after the execution of the Process, the condition is checked to be True or False. When the condition is found to be False the program control comes out of the loop. This condition checking can be carried out at the end of the loop also. All these situations are possible in C using the **while, do-while** and **for** loop statements.

When a set of instructions get repeated several times depending on a condition, it forms a Loop

• The Increment, Decrement, and Compound Assignment Operators:

Before we learn about the different looping processes let us learn about some operators, apart from the relational operators, that are used extensively in programs related to iteration. The table below shows a set of these operators called **increment, decrement and compound assignment operators**.

Operator	Description	Examples	Remarks

++	Increments the existing value of a variable by 1.	`a++;`
		`++a;`
		`y = a++;`
		`x = ++z+6;`
			Post increment `x++` is same as `x=x+1`
			Pre increment `++x` is same as `x=x+1`

--	Decrements the existing value of a variable by 1.	`a--;`
		`--a;`
		`y=a--;`
		`x=--z+6;`
			Post decrement `x--` is same as `x=x-1`
			Pre decrement `--x` is same as `x=x-1`

+=	Increases the existing value of a variable to the left of the operator by an amount to the right.	`y+=5.2;`
		`z+=2+3*y;`
		`y+=1;`
			`y+=5.2` same as `y=y+5.2`
			`z+=2+3*y` same as `z=z+(2+3*y)`
			`y+=1` same as `y=y+1` same as `y++`

-=	Decreases the existing value of a variable to the left of the operator by an amount to the right	`y-=4.5;`
		`z-=2*b+c;`
		`y-=1;`
			`y-=4.5` same as `y=y-4.5`
			`z-=2*b+c` same as `z=z-(2*b+c)`
			`y-=1` same as `y=y-1` same as `y--`

*=	Multiplies the existing value of a variable to the left of the operator by an amount to the right	`y*=k;`
		`z*=5+3/y;`
		`y*=1;`
			`y*=k` same as `y=y*k`
			`z*=5+3/y` same as `z=z*(5+3/y)`
			`y*=1` same as `y=y*1`

/=	Divides the existing value of a variable on the left of the operator by an amount on the right	`y/=k;`
		`z/=x+3*y;`
		`p/=2;`
			`y/=k` same as `y=y/k`
			`z/=x+3*y` same as `z=z/(x+3*y)`
			`p/=2` same as `p=p/2`

%=	Calculates the mod of an existing value of a variable to the left of the operator by an amount to the right	`a%=5*t;`
		`z%=5%a+4;`
		`y%=7;`
			`a%=5*t` same as `a=a%(5*t)`
			`z%=5%a+4` same as `z=z%(5%a+4)`
			`y%=1` same as `y=y%7`

Increment, Decrement, and Compound operators

The increment operator ++ is the most widely used operators in a loop. It increases the value of a variable bu 1

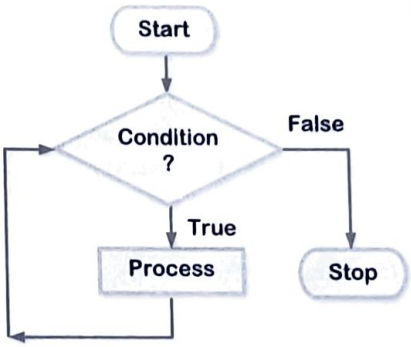
CHAPTER 11
Using Loops in C

- Introduction 11-1
- Concept of a Loop 11-2
- The while Loop 11-6
- The break statement 11-14
- The continue statement 11-15
- The do-while Loop 11-16
- The for Loop 11-18
- Some worked out examples 11-24

11.1 Introduction

Apart from conditional branching there may be situations where a particular set of instructions may need to be repeated several times depending upon some conditions until the desired result is got.

As an example take the situation of finding the terms of an A.P. series up to certain number of terms. Each new term is created by adding the common difference to the previous term. This forms a repetitive process. Such a repetitive process is also called **iteration**. The number of times to repeat usually depends upon conditions.



Flowchart for looping

When a set of instructions get repeated several times depending on a condition, it forms a Loop

The general logic for such a situation is given by the **flowchart** on the right. As long as the condition is True, the **Process part is repeated**. Each time after the execution of the Process, the condition is checked to be True or False. When the condition is found to be False the program control comes out of the loop. This condition checking can be carried out at the end of the loop also. All these situations are possible in C using the **while**, **do-while** and **for** loop statements.

• The Increment, Decrement, and Compound Assignment Operators:

Before we learn about the different looping processes let us learn about some operators, apart from the relational operators, that are used extensively in programs related to iteration. The table below shows a set of these operators called **increment, decrement and compound assignment operators**.

Operator	Description	Examples	Remarks

++	Increments the existing value of a variable by 1.	`a++;`
		`++a;`
		`y = a++;`
		`x = ++z+6;`
			Post increment `x++` is same as `x=x+1`
			Pre increment `++x` is same as `x=x+1`

--	Decrements the existing value of a variable by 1.	`a--;`
		`--a;`
		`y=a--;`
		`x=--z+6;`
			Post decrement `x--` is same as `x=x-1`
			Pre decrement `--x` is same as `x=x-1`

+=	Increases the existing value of a variable to the left of the operator by an amount to the right.	`y+=5.2;`
		`z+=2+3*y;`
		`y+=1;`
			`y+=5.2` same as `y=y+5.2`
			`z+=2+3*y` same as `z=z+(2+3*y)`
			`y+=1` same as `y=y+1` same as `y++`

-=	Decreases the existing value of a variable to the left of the operator by an amount to the right	`y-=4.5;`
		`z-=2*b+c;`
		`y-=1;`
			`y-=4.5` same as `y=y-4.5`
			`z-=2*b+c` same as `z=z-(2*b+c)`
			`y-=1` same as `y=y-1` same as `y--`

*=	Multiplies the existing value of a variable to the left of the operator by an amount to the right	`y*=k;`
		`z*=5+3/y;`
		`y*=1;`
			`y*=k` same as `y=y*k`
			`z*=5+3/y` same as `z=z*(5+3/y)`
			`y*=1` same as `y=y*1`

/=	Divides the existing value of a variable on the left of the operator by an amount on the right	`y/=k;`
		`z/=x+3*y;`
		`p/=2;`
			`y/=k` same as `y=y/k`
			`z/=x+3*y` same as `z=z/(x+3*y)`
			`p/=2` same as `p=p/2`

%=	Calculates the mod of an existing value of a variable to the left of the operator by an amount to the right	`a%=5*t;`
		`z%=5%a+4;`
		`y%=7;`
			`a%=5*t` same as `a=a%(5*t)`
			`z%=5%a+4` same as `z=z%(5%a+4)`
			`y%=1` same as `y=y%7`

Increment, Decrement, and Compound operators

The increment operator ++ is the most widely used operators in a loop. It increases the value of a variable bu 1

The **increment operator when used separately does the same job as incrementing the variable to which it is attached by 1.** It is usually used in loop applications to increment a counter variable by 1, each time a certain portion of the loop is executed.



Concept of a loop

11.2 Concept of a loop

Suppose we want to find the average of 3 numbers entered by the user. With the programming knowledge we have so far, we can do it using the following code:

```
1  /*Program-41: Program to find average of three numbers*/
2  #include<stdio.h>
3  int main()
4  {float num1, num2, num2, sum=0, average;
5   printf("\nEnter number: ");
6   scanf("%d", &num1);
7   sum = sum + num1;                    /*Adds num1 to the last value of sum i.e. 0*/
8   printf("\nEnter number: ");
9   scanf("%d", &num2);
10  sum = sum + num2;                    /*Adds num2 to the last value of sum i.e. num1*/
11  printf("\nEnter number: ");
12  scanf("%d", &num3);
13  sum = sum + num3;                    /*Adds num3 to the last value of sum i.e. num1+num2*/
14  average = sum/3;
15  printf("\nAverage of the numbers is %.2f", average);
16  return 0;
17 }
```

Output:

```
Enter number: 5.2
Enter number: 7.9
Enter number: 12.5
Average of the numbers is 8.53
```

In the above program the following three operations are repeated to input the numbers and get the sum:

```
5   Display a prompt to input the number
6   Input the number
7   Add the input number to the existing sum
```

In line-4, the variable **sum** is initialised to **0**. In line-7, the code **sum=sum+num1** adds the input value **num1** to the initial value of **sum** i.e. **0** to get the value **0+num1=num1**. It then assigns this value to the variable **sum**. The last value in **sum**, i.e. **0**, thereafter gets overwritten by the new value **num1**. So **sum** contains the value **num1** after line-7.

In line-10, again the new value in **num2** is added to the last value stored in **sum** i.e. **num1** to get the value **num1+num2**. It then assigns this value to the variable **sum**. The last value in **sum**, i.e. **num1**, therefore gets overwritten by the new value **num1+num2**. So **sum** contains the value **num1+num2** after line-10.

Finally in line-13 the new value in **num3** is added to the old value in **sum** i.e. **num1+num2** to get the value **num1+num2+num3**. It then again assigns this value to the variable **sum**. The last value in **sum**, i.e. **num1+num2**, therefore gets overwritten by the new value **num1+num2+num3**. So **sum** contains the value **num1+num2+num3** after line-13.

Line-14 then calculates the **average** by dividing the final value in **sum** i.e. (**num1+num2+num3**) by 3.

There are **two problems** with this approach:

- If the **number of values to enter is large** (for example the average of 1000 numbers), we have to write the same code many times. We can use copy-paste option to replicate the code. But it will unnecessarily use extra time and memory and at the same time the program listing will also be too long.

- In case the **number of times to repeat the code is not known beforehand** and depends upon one or more conditions or is an user input, then we will not know how many times to replicate the same code during writing the program.

The following pseudo-code is an **alternative approach** to the same program using the **concept of loops** that can overcome the above problems.



Pseudocode for a loop

```
/*Program-41a: Pseudocode to find average of three numbers using loop*/
#include<stdio.h>
int main()
{int count=1; float num, sum=0, average;
   repeat lines 6 to 9 as long as value in count is less than or equal to 3
     printf("\nEnter number: ");
      scanf("%d", &num);
      sum = sum + num;
      count = count + 1;
   end_of_loop
average = sum/count;
printf("\nAverage of %d numbers is %.2f", count, average);
return 0;
}
```

Output:

```
Enter number: 5.2
Enter number: 7.9
Enter number: 12.5
Average of the numbers is 8.53
```

Usually a counter is used to count the number of times the loop has run.

In the above code, the variables **sum** and **count** have been initialised to **0** and **1** respectively during the declaration in line-4. The variable **sum** stores the result of adding the 3 numbers while the variable **count** is used as a counter to keep track of the number of values added. In programs dealing with loops, often **we will be using a counter in this manner to count the number of iterations** i.e. to count the number of times the loop has repeated.

As we want to add 3 numbers, the loop is required to repeat 3 times. To do that the counter has been initialised to 1. Each time the loop portion from line-5 to line-9 is run, the counter is increased in value by 1 by the statement **count=count+1** in line-9. Finally when the value of count is equal to 3, the loop portion is repeated for the last time. After that the condition in line-5 becomes False and the program control straight away goes to line-11 where the average is calculated. The manner in which the values in the different variables change is shown in the table above.

Values in the variables during the execution of the code

Repeat	count	Condition count <= 3	num	sum	Remark
1	1	True	5.2	0+**5.2**=5.2	count increased by 1
2	2	True	7.9	5.2+**7.9**=13.1	count increased by 1
3	3	True	12.5	13.1+**12.5**=**25.6**	count increased by 1
4	4	False			Loop stops

Note how the three repeating lines of program-40 (lines-5, 6, 7) have been used in program 41a. Also, a single variable **num** has been used to input all the three numbers, instead of three variable **num1**, **num2**, and **num3**. Each time the loop is run, a value is entered in line-7 in the variable **num** using the **scanf()** function. The value is then added to the existing value of the variable **sum** in line-8. So, when the first value 5.2 is entered and added to the existing value of **sum** i.e. 0 in line-8, **sum** contains the new value 0+5.2 i.e. 5.2. Next the variable **count** is incremented by 1 in line-9. So **count** now contains the value 1+1=2. In this way the code continues to function.

After the end of the last repeating line when the **end_of_loop** is reached, the program **control goes back to the condition part** in line-5. There the condition is checked to see if it is True or False. If the condition is True the part from line-6 to 9 is again repeated.

During subsequent repetition of the loop new numbers will be input into the same variable **num** (overwriting the last input value) and added to the existing **sum** to get a new **sum** value. The value of **count** also gets incremented by 1 every time. Finally when the result of the condition checking in line-5 becomes False, the loop terminates and the control goes to the next line outside the loop. Once outside the loop, the final **sum** is divided by **count** to get the **average** in line-11.

• The Increment and Decrement Operators as a statement:

The increment and decrement operators ++ and -- are extensively used in programs involving iteration or loops as stated earlier. These can be placed at two different places with respect to the variable, depending upon the situation. These are:

Pre and Post Increment operation

- **Pre increment/decrement**: When the operator is placed before a variable like ++x, or --x
- **Post increment/decrement**: When the operator is placed after the variable like x++, or x--

When you use the increment/decrement operators to increase/decrease the value of a variable you can use it in the following manner:

++x is called a pre-increment operation.
x++ is called a post-increment operation.

```
1  int x=1;
2  printf("%d", x);      /*Prints the value of x as 1*/
3  ++x;                 /*Increments the value of x by 1 i.e. x=x+1. So x becomes 2*/
4  printf("\n%d", x);    /*Prints the value of x as 2*/
5  x++;                 /*Increments the value of x by 1 i.e. x=x+1. So x becomes 3*/
6  printf("\n%d", x);    /*Prints the value of x as 3*/
```

Output:

```
1
2
3
```

When used in the above manner, the **final result is the same** as is evident in the above output. The same rule applies for the decrement operators also.

Note that we can **replace** the operators in lines-8 and 9 in the last program (program-41a) by the compound and increment operators as shown below:

```
8  sum += num;   ⇒ instead of sum = sum + num;
9  ++count;      ⇒ instead of count = count + 1;
```

However, when the operators are used in an expression, the outputs of the pre-increment/decrement and post-increment/decrement operations may differ.

• The Increment and Decrement Operators in an expression or condition:

Increment operators in an expression

When using the increment (++) and decrement (--) operators in an expression or a condition, **there may be a difference** between the pre-increment/decrement and post-increment/decrement operations. The following codes illustrate the difference between the two. When you write:

```
int a=5, num=0;
num = ++a;
```

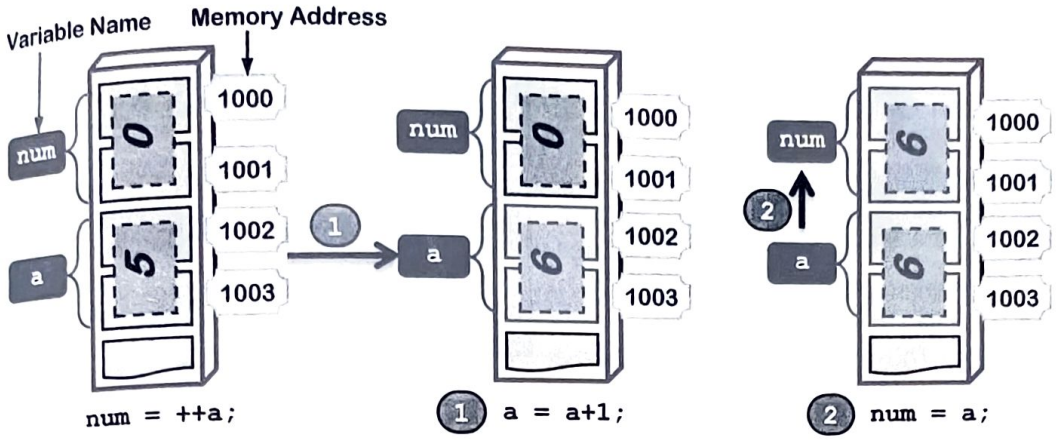
y=++x; is same as:
x=x+1;
y=x;
y=x++; is same as:
y=x;
x=x+1;

The two variables **a** and **num** are first initialised to 5 and 0 respectively. In the second line we have used the **pre-increment** form of the operator in the expression.

When **pre-increment** takes place in an expression, the variable is **first incremented** and then the **incremented value is used**. The variable **a** is thus first incremented by 1 and it assumes the value 5+1=6. **After** the increment, it is assigned to the variable **num**. Thereafter the variable **num** also gets the value 6. This is illustrated by the memory diagram shown in the next page. The statement **num = ++a;** is thus equivalent to the two statements in this particular order:

```
1  a = a+1;
2  num = a;
```

} Same as: num = ++a;



Meaning of num=++a;

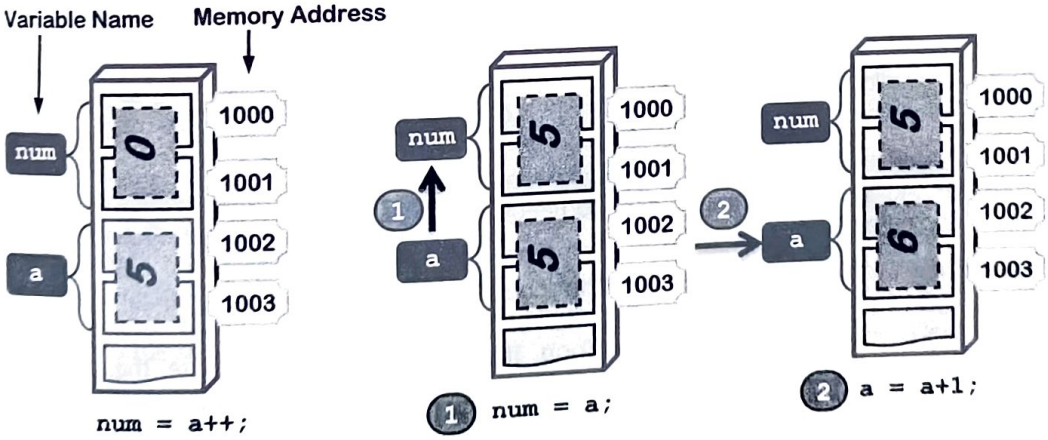
Now let us find out what happens when we write the same code but use the post-increment operation.

```
int a=5, num=0;
num = a++;
```

The two variables `a` and `num` are first initialised to 5 and 0 as before. In the second line we have used the **post-increment** form of the operator in the expression.

When **post-increment** takes place in an expression, the variable is **first used** and **then incremented**. As in the above code only assignment operation takes place, the variable `a` is first assigned to the variable `num` and thereafter `num` assumes the value 5. **After** the assignment, the variable `a` is incremented by 1 and it gets the value 5+1=6. Therefore the variable `a` now stores the value 6. The operation is illustrated by the memory diagram shown below. The statement `num = a++;` is thus equivalent to the two statements in this particular order:

```
1  num = a;
2  a = a+1;
```
Same as: `num = a++;`



Meaning of num=a++;

Note: There are **no operators as** `+++` or `---` in C. Hence `+++x`, `x+++`, or `y---`, `---y` does not have any meaning by itself, though, `z=x+++y` does carry a meaning and is the same as `z=(x++) + y`.

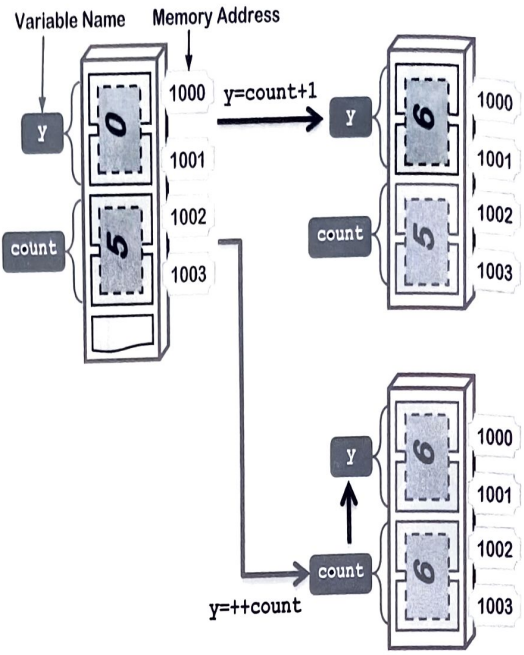
Difference between the expressions count+1 and ++count:

Difference between a+1 and ++a

While using the increment/decrement operators, one more thing should be noted. **There is a difference between the following two statements** though the effect of these statements on the variable `y` is the same.

```
y = count+1;
y = ++count;
```

The diagram on the right shows the difference for the initial value of `count` taken as 5 and `y` taken as 0. It can be seen that though the **final content** of `y=count+1` and `y=++count` are **same** for a particular initial value of `count` (in this case 5) but the way the two statements function are different.



In the first statement the value of `y` is derived by adding `1` to the existing value of `count`. Thus `y` becomes equal to `6` whereas `count` remains at 5.

But in the second statement, the code `++count` first increments the value of `count` by `1` and then assigns the increased value of `count` to `y`. Thus first `count` becomes equal to `6` and then this value is assigned to `y` which then becomes equal to `6`. Therefore:

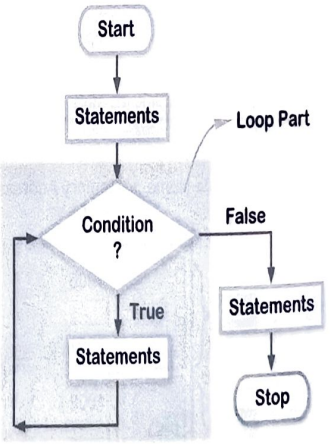
`y=count+1` does not change the value of count, but changes the value of y.
`y=++count` changes the value of both count and y.

```
printf("New count=%d", count+1);   ⇒ Displays count+1, with no change in value of count
printf("New count=%d", ++count);   ⇒ Changes value of count by 1 and then displays the result
```

11.3 The while Loop

In this section we will discuss how we can repeat a set of program statements using a **`while` loop**.

The flowchart on the right shows the logic of the `while` loop. The condition part of the loop may be constructed using relational operators (similar to the condition in an if-else statement) or using some other logic. The condition part may check the value of a counter variable to decide how many times to repeat the code. However the use of a counter is not compulsory and other logics may be used in the condition part to keep a loop running or terminate a loop.



The while loop is an entry controlled loop, which tests the condition first before entering the loop body.

In case a counter variable is used, it needs to be declared and initialised outside the loop. Within the loop it needs to be incremented or decremented as per need.

The shaded portion in the diagram indicates the loop. Each time the loop begins to execute, **the condition is tested first** and if found True, the body of the loop is executed. It is thus called an **entry controlled loop**. If the condition becomes False, the control goes out of the loop to the statement after the loop. The general structure of the `while` loop is shown below:

```
Initialise_loop_counter;   /*optional*/
while (condition)
      { statements;
        ..... .....
        change_loop_counter; /*optional*/
      }
```

If True, execute body

No semicolon after the while keyword

Body of the loop indicated by enclosing them within the curly braces { }

Note that there is **no semicolon** after the `while` condition. In case the condition becomes True then the statements in the body of the loop gets executed. The statements to be executed as part of the loop are indicated by **enclosing them within a pair of curly brackets i.e. { }**.

Let us rewrite the **program 41a** to find the average of three numbers using a `while` loop.

```
1  /*Program-41b: Program to find average of three numbers using while loop*/
2  #include<stdio.h>
3  int main()
4  {int count=1; float num, sum=0, average;
5   while (count<=3)
6         {printf("\nEnter number: ");
7          scanf("%d", &num);
8          sum = sum + num;
9          count = count + 1;
10        }
11  average = sum/count;
12  printf("\nAverage of %d numbers is %.2f", count, average);
13  return 0;
14 }
```

Program to find average of 3 numbers

Note how the condition part is written using relational operators in line-5. The body of the loop that is repeated is from line-6 to line-9. The opening and closing curly braces in line-6 and line-10 enclose the body of the loop. We have used a counter variable called **count** to count the number of times the loop has run.

Let us take a simple example to illustrate the functioning of the **while** loop. The following program prints the numbers from 1 to 5.

```
1  /*Program-42: Use of while loop to print from 1 to 5*/
2  #include<stdio.h>
3  int main()
4  { int count=1;
5    while(count <= 5)
6      { printf("\nNumber %d", count);
7        ++count;
8      }
9    return 0;
10 }
```

When last line in the body of the loop is reached, control goes back to loop header to recheck condition

Output:

```
Number 1
Number 2
Number 3
Number 4
Number 5
```

In the above program, the **number of times the loop should repeat** is checked by the variable **count** (you can give any other suitable name). The program functions in the following way:

a) In line-4 during the declaration part the variable **count** is **initialised to 1**

b) In line-5 the loop control structure begins and the **condition following the while keyword is checked**. In this case it is checked if **count<=5**

c) **If the condition is satisfied or evaluates to True**, the body of the **while** loop is executed.
   i. Once within the body of the loop the **printf()** statement in line-6 prints the value of **count**
   ii. The variable **count** is then incremented in line-7 by using the **increment operator ++**
   iii. As the program control reaches the last line in the body of the loop, it again goes back to line-5 and rechecks the condition. If it is found to be True the control again enters the body of the loop in line-6

d) If the condition is found False then the body of the loop is skipped and the statement immediately after the body of the loop i.e. in this case **return 0** in line-9 gets executed.

Working of while loop

When the loop is run for the first time, the loop condition is satisfied as **count=1**, is less than 5. The program control enters the body of the loop and prints the value of **count**. Next it increments the value of **count** by 1 in **++count** and **count** becomes equal to 2. On reaching the end of the loop block, the

program again checks if the condition is satisfied. Since `count` is now equal to `2`, the condition is again satisfied and the block of statements inside the loop again gets executed. This continues till `++count` makes `count` equal to `5`. This is the last time the loop will be executed, for after this `count` becomes equal to `6` which fails the condition and the loop terminates.



Different ways of using the condition in a while loop

The above program can be written **in a different manner also** as indicated below:

```
1   int count=0;
2   while (++count <= 5)
3       printf("\nNumber %d", count);
```

The variable `count` **has been initialised to 0** instead of **1** in line-1. On entering the `while` loop condition part, `count` is first incremented to **1** by `++count` (pre-increment) and then the condition is tested. The condition will remain True till `++count` makes `count=5` when the loop will be executed for the last time and will print `Number 5`. Note that the curly braces for the body of the loop are also not used as we have a **single statement to execute** as part of the loop.

In the previous example we have used the **pre-increment operator** as `++count`. If we were to use the **post-increment** operator as `count++` then the program will have to be changed in the following way:

```
1     int count=0;
2     while (count++ < 5)
3         printf("\nNumber %d", count);
```

In line-2 we have used the **post-increment** operator as `count++`. Therefore the increment will occur after checking the condition. When the condition will be tested for the first time, `count` will be equal to `0` which will satisfy the condition `count<5`. After testing the condition `count` will be incremented to `1` by `count++`. Thus the value of `count` that will be printed for the first time in line-3 will be `1` and not `0`. Note that the equal-to sign has been removed and only < is used inside the condition. This is because to print `5` as the last number in line-3, `count` will have to be equal to `4` during testing of the condition, so that the `++` after the testing makes it equal to `5`. If <= were used then in the next run of the loop the condition would have been satisfied and the output would have been `6` after the `++` operation on `count` in line-2.

To find the **factorial of a number** using a `while` loop.



Factorial of a number using a while loop

```
1   /*Program-43: To find the factorial of a number*/
2   #include<stdio.h>
3   int main()
4   { int num, count=1;
5     long int fact=1;
6     printf("\nEnter number to find factorial: ");
7     scanf("%d", &num);
8     while( count <= num )
9       { fact*=count;              /*  Same as fact = fact*count;  */
10        ++count;
11      }
12    printf("\nThe required factorial of %d is %ld", num, fact);
13    return 0;
14  }
```

For series product initialise the product variable to 1 always.

When calculating a factorial value take the factorial variable as a long int or float for better results

Output:

```
Enter number to find factorial: 5
The required factorial of 5 is 120
```

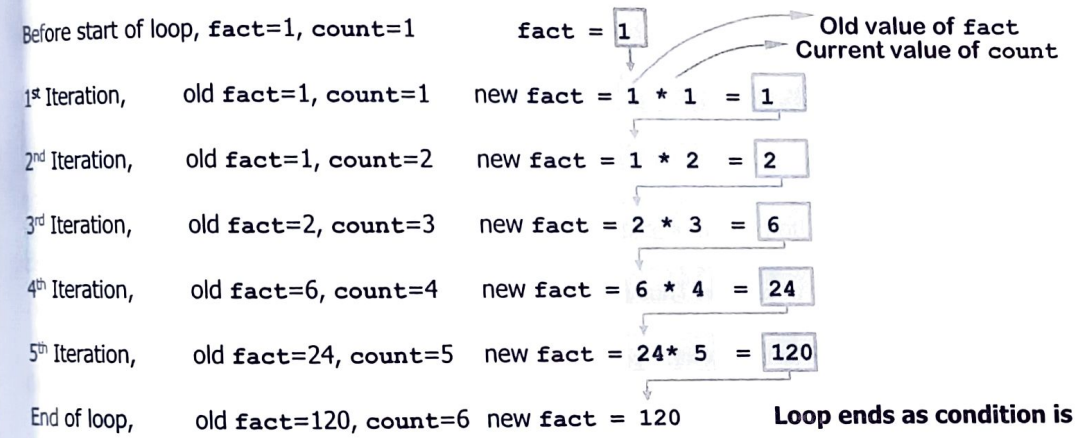
The variable `num` is used to input the number whose factorial is to be found. The variable `count` is used to **count the number of times the loop has executed** and hence has been initialised to `1`.

The variable `fact` **is declared as a** `long int` type data in line-5 and used for **storing the value of the factorial**. This is essential as the value of factorial increases very rapidly and even factorial of 8 is larger than the range of a two byte integer. Hence `fact` has been taken as a `long int`.

The **variable `fact` has been initialised to `1`** and **not to `0`** in line-5. **Initialisation is a must**, otherwise `fact*=count` or `fact=fact*count` will give an unpredictable value when used for the first time as the new value of `fact` is calculated from the old value of `fact` by multiplying it by `count`. (Remember that an un-initialised variable contains garbage or unpredictable value).

The loop starts from line-8. The **condition remains True as long as the value of the variable `count` remains less than or equal to `num`**. To calculate the factorial of `num`, the loop has to run `num` number of times. For example to calculate the factorial of `5`, the loop has to run `5` times. When the loop runs for the first time, `count=1` and the condition satisfies in line-8. Within the loop, the value of `count` i.e. `1` is multiplied with the value of `fact` i.e. `1` to get the new value of `fact` as `fact=1x1`.

The counter is then incremented to `2` in line-10. The loop condition again satisfies. Within the loop, `count` i.e. `2` gets multiplied with old `fact` to get the new value of `fact` as `fact=1x2=2`. `count` is next incremented to `3` in line-10. The loop condition again satisfies and now the new value of `fact` becomes equal to `fact=2x3=6`. `count` is incremented to `4` and in the next iteration new value of `fact` becomes equal to `fact=6x4=24`. This process continues till `count` becomes equal to `5` in the next step. The condition satisfies for the last time and the new value of `fact` is calculated as `fact=24x5=120`, which is the value of factorial of `5`. The following diagram shows the procedure.



Logic behind finding factorial

Note that the value of old `fact` is always derived from the previous iteration.

We give below two more variations of the above program.

```
1    count=1;                    /*Second version of factorial program*/
2    while( count <= num )
3       fact = fact*count++;
```

Different ways to find factorial

In the above code, the variable count is incremented in the same line where it is used. This is done using the post-increment operator. After the product `fact*count` is made in line-3, the **post-increment operator** increments the value of `count` by 1 in the same line.

```
1    while( num>0 )                 /*Third version of factorial program*/
2       fact = fact*num--;
```

In the above code we have not used the `count` variable but used the variable `num` as a counter. For `num=5`, the `while` condition is True when the loop starts. After the product `fact*num` is made in line-2, the **post-decrement** operator decreases the value of `num` by `1` in the same line (`num--`). So `num` now becomes equal to `4`. The `while` condition is found to be **True** with `num=4` and the loop body again executes. The process continues till the value of `num` is equal to `1`. Line-2 runs for the last time. After that `num--` decrements the value of `num` to `0`. When the `while` condition is checked next, the condition `num>0` i.e. `0>0` becomes **False** and the loop terminates.

We give below a final version of the loop portion for the above program. Try to analyse the program on your own with the knowledge you have gained so far!

```
1    while( num>0 && fact*=num-- );  /*Fourth version of factorial program*/
```

The next program is used to find the value of x^y, where x is a float type value and y is a whole number.

To find xʸ

To find the **value of x^y**, using a while loop where x and y are entered by the user.

```
1   /*Program-44: To find the power of a number using a while loop*/
2   #include<stdio.h>
3   int main()
4   { float x; int y, count=1;
5     float power=1;
6     printf("\nEnter base value: ");
7     scanf("%f", &x);
8     printf("\nEnter index value: ");
9     scanf("%d", &y);
10    while( count <= y )
11      { power = power*x;
12        ++count;
13      }
14    printf("\nThe required power is %f", power);
15    return 0;
16  }
```

Output:

```
Enter base value: 4.3
Enter index value: 4
The required power is 17.2
```

In this case the loop should run **y** number of times as x^y means **x** multiplied **y** number of times. Like the previous problem, with each iteration **x** is multiplied with the old value of the variable **power** to get the new value of **power**. For example for **x=4** and **y=3**, the loop should run 3 times. Each time the value 4 gets multiplied with the previous value of **power**. Thus the steps involved are:

Old value of power

Value of x

Before start of loop, **power**=1, **count**=1, x=4: **power** = 1

1st Iteration, old **power**=1, **count**=1 new **power** = **power * x** = 1 * 4 = 4

2nd Iteration, old **power** =4, **count**=2 new **power** = **power * x** = 4 * 4 = 16

3rd Iteration, old **power** =16, **count**=3 new **power** = **power * x** = 16* 4 = 64

4th Iteration, old **power** =64, **count**=4 new **power** = 64 **Loop ends as condition is False**

We give below another **variation of the above program** which does not use the counter variable **count**. The logic of operation is similar to the previous one.

```
while(y--)
   power = power*x;
```

The last two programs are examples of **series products** where the final product is derived by successively multiplying the starting product with a variable or a constant term using a code like `product=product*term;`. Like a series product you can also have a **series sum**, where the final sum is derived by successively adding the starting sum value with a variable or a constant term using a code like `sum=sum+term;`

In both cases you **must initialise** the starting product or sum value. Usually the **product is initialised to 1** and the **sum is initialised to 0** (exceptions can be there as per specific programming needs).

While loop without using a proper counter

- Using while loop without the use of a proper counter variable:

It is not always necessary to use a counter to keep a count of the number of times the loop has run. The value of a suitable variable can be used to keep track of the number of times the loop has run and terminate the loop accordingly. The concept can be shown with the help of a set of **problems that deal with the individual digits of an integer number**.

As an example, the following program finds the **sum of the digits of any number** entered by the user using a while loop. To do this, the digits of a number are extracted one by one from the right of the number and then added to get the final sum. The modulo operator (%) is used to extract the digits of the number as shown in the program in the next page.

Note that in getting the sum, **three basic steps (i to iii) are repeated**. The algorithm for this is:

a. Check if the input number **num** is non-zero (for positive numbers). If so,
   i. Get the rightmost **digit** of the number **num** by using **remainder division** operation (num%10)
   ii. Use the **digit** obtained in step (i) as per the requirement of the problem (e.g. to get the sum)
   iii. Reduce the number **num** by truncating the rightmost digit of the number using **integer division** (num/10)
   iv. Get back to step (a)



Algorithm to extract the digits of a number



Finding sum of digits of a number

```
/*Program-45: Getting the sum of the digits of a number*/
#include<stdio.h>
int main()
{ long int num;
  int digit, sum=0;
  printf("\nEnter number to add digits: ");
  scanf("%ld", &num);
  while( num > 0 )
    {digit = num%10;
     sum = sum + digit;
     num = num/10;
    }
  printf("\nThe sum of the digits of the number is %d", sum);
  return 0;
}
```

For doing a series sum, always initialise the sum variable to 0.

Output:

```
Enter number to add digits: 258
The sum of the digits of the number is 15
```

Let us analyse the above program with the input value **258**, to understand the process:

- In line-7 the value to check is entered into the variable **num**. In our example, **258** is entered in **num**
- The loop starting from line-8 is used to **extract each digit from the number** and add it to get the sum. As **258 > 0**, the **while** condition is satisfied and the program enters the **while** loop
- In line-9 the variable **digit = num%10 = 258%10 = 8** is obtained (remainder of **258** divided by **10**)
- In line-10 new **sum = sum + digit = 0 + 8 = 8** (as **sum** had been initialised to **0** in line-5)
- In line-11, the new value of **num = num/10 = 258/10 = 25** is obtained (integer division of **258** by **10**)
- After the last line inside the loop, the control goes back to line-8, and rechecks the condition. The condition is True as the new value of **num** is **25** and **25>0** is **True**.
- In line-9, **digit = num%10 = 25%10 = 5** is obtained (remainder of **25** divided by **10**)
- In line-10, new **sum = sum + digit = 8 + 5 = 13** (as old value of **sum=8** from the last iteration)
- In line-11, the new value of **num = num/10 = 25/10 = 2** (result of integer division of **25** by **10**)
- Control again goes back to line-8, where the condition is True as new value of **num=2** and **2>0** is True
- In line-9 **digit = num%10 = 2%10 = 2** is obtained (remainder of **2** divided by **10**)
- In line-10, new **sum = sum + digit = 13 + 2 = 15** (as old value of **sum=13** from the last iteration)
- In line-11, the new value of **num = num/10 = 2/10 = 0** (result of integer division of **2** and **10**)
- Control goes back to line-8, where condition becomes **False** as new value of **num=0** and **0>0** is False

Thus the final value of sum obtained is equal to the sum of the digits of the number 258 i.e. 8+5+2=15.

The **while** loop is more suited for **execution of a loop unknown number of times** depending on certain calculations or user input. In the last example, the loop continues as long as the number is non-zero depending upon the number of digits in the input value. The loop repeats different number of times for different number of digits in the value checked.

Another such example is to repeat a certain portion of code depending on whether the user wants to repeat it, as demonstrated in the program below (though such code is better suited for a **do-while** loop).

```
1   /*Program-46: Repeating a program*/
2   #include<stdio.h>
3   int main()
4   { char repeat='y';
5     float num;
6     while( repeat=='y' || repeat=='Y' )
7        { printf("\nEnter any decimal number to square: ");
8          scanf("%f", &num);
9          printf("\nThe square of %f is %f", num, num*num);
10         puts("\nPress 'Y' to repeat with a new number or any other key to exit.");
11         fflush(stdin);
12         repeat = getchar();
13       }
14    return 0;
15  }
```

Output:

```
Enter any decimal number to square:  6.2
The square of 6.2 is 38.440000
Press 'Y' to repeat with a new number or any other key to exit.
Y
Enter any decimal number to square:  5.9
The square of 5.9 is 34.810000
Press 'Y' to repeat with a new number or any other key to exit.
e
```

The above program is used to calculate the square of a number as many times as the user wants, without the need to restart the program each time. The condition of the **while** loop checks the value of a character type variable called **repeat**. In case the value of **repeat** is '**y**' or '**Y**', the condition of the loop gets satisfied. The variable **repeat** is first initialised to '**y**' in line-4 to enter the loop for the first time. Subsequent runs of the program depend on the input of the user to the prompt to repeat in line-10.

As long as the user types '**y**' or '**Y**', the **while** condition gets satisfied and repeats the section of the code from line-7 to line-13. (Note that the logical OR operator has been used inside the **while** condition to test if the user has entered '**Y**' or '**y**', for both of which the condition should be True). The user prompt is entered using the **getchar()** function in line-12. Note that the **fflush(stdin)** function has been used in line-11.

Note: Instead of a character type prompt, one can use an integer type prompt with values 0 and 1.

A while loop may not have a proper loop body and the loop header may end with a semicolon.

You can also have a **valid while loop without a proper loop body** as shown below:

```
1   int x=6;
2   while( x-- );
3   printf("x = %d", x);
```

In the above code note that a **semicolon has been placed** after the **while** header. This implies that there is no loop body and the loop ends at line-2 itself. The loop header functions as the loop body and the **x--** code decrements at each run of the code till it becomes zero, when the loop stops (as 0 is taken as False).

Nested loops

- Using nested while loops:

When one loop control structure is placed within another one, then the inner loop is called a nested loop. The following example prints a simple right angled triangular pattern on screen using the concept of a nested loop and based on the number of lines to print as input by the user. For the number of lines as 5, the program should print:

```
*
**
***
****
*****
```

Program to print simple triangle pattern

```
/*Program-47: Use of nested while loops to print pattern*/
#include<stdio.h>
int main()
{ int n, count=1, j;
  printf("\nEnter the number of lines to print: ");
  scanf("%d", &n);
  while(count <= n)          /*This loop is used to count the number of lines*/
   { j = 1;                  /*Star counter j is initialised to 1 for every new line*/
     while( j <= count )     /*This loop is used to print the number of stars in a line*/
      { printf("*");
        ++j;                 /*Counter for number of stars incremented*/
      }
     printf("\n");           /*printf() prints a new line i.e. moves to next line*/
     ++count;                /*Counter for number of lines incremented*/
   }
  return 0;
}
```

Output:

```
Enter the number of lines to print:  5
*
**
***
****
*****
```

Line-6 enters the number of lines of the pattern to print. Let us analyse the program for number of lines as 5. The **count** variable declared in line-4 is initialised to 1. It will be used to count the number of lines printed.

The outer **while** loop in line-7 is used to count the number of lines printed and checks the condition to see if the required number of lines have been printed or not. To start with, as **count=1**, and **n=5**, the condition is True and the control enters the outer **while** loop.

Within the outer loop in line-8 a variable **j** is initialised to 1. It counts the number of stars printed in a given line. The inner **while** loop of line-9 is used to print the number of stars depending upon the line number.

When **count=1**, and **j=1**, the inner **while** loop condition **j<=count** is True and the **printf()** statement of line-10 prints a star, the variable **j** is next incremented to 2 in line-11. With **j=2**, the condition of the inner while loop becomes False and the inner loop terminates. The **printf()** statement of line-13 then prints a new line to move to the next line. The line counter **count** is thereafter increased by 1 in line-14.

The program control goes back to check the condition of the outer **while** loop. With **count=2** and **n=5**, the condition is found to be True and the control enters the outer **while** loop again.

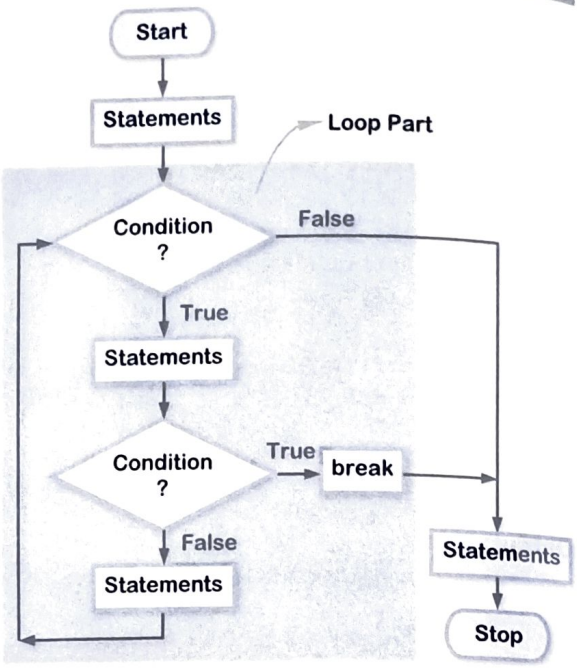
The star count variable **j** is again initialised to 1 for the second line of stars. With **count=2** and **j=1**, the inner **while** loop condition becomes True and the **printf()** statement of line-13 prints a star in the second line. The variable **j** is incremented to 2 in line-10 and the inner loop condition checked again. With **count=2**, and **j=2** the condition is True again and the **printf()** statement prints the second star (*) in the second line. The variable **j** is incremented to 3 next. With **count=2** and **j=3**, the inner **while** loop condition becomes False and the inner loop terminates. The **printf()** statement of line-13 prints a new line. The variable **count** is incremented to 3 next and the outer loop condition rechecked.

In this way for the first 5 runs of the loop, the variable **count** will get increment by 1 and have the values 1, 2, 3, 4, 5. For each of these values, the inner **while** loop will run that many number of times to print the stars. Thus when **count=3**, the inner loop will print 3 stars, when **count=4**, the inner while loop will print 4 stars and so on. After printing every line of stars, the **printf()** statement of line-13 will introduce a new line so that the next line of stars starts printing from the next line.

Note that the variable **j** has been initialised within the outer loop. The reason for this is that, every time the inner **while** loop runs, the variable **j** gets changed and needs to be initialised to 1 again for the next operation of the outer loop.

11.4 The break statement

The `break` statement is used **to forcibly come out of a program loop bypassing the normal loop condition test** or to terminate a `case` in the `switch` statement (as discussed in the last chapter). The program control comes out of the loop on the execution of a break statement and the statement immediately next to the loop gets executed.



The flowchart on the next page explains the working of the `break` statement. The condition to break out of a loop is generally checked using an `if` statement. Usually if the condition becomes True, the control comes out of the loop and if it is found to be False, it continues to execute the loop.

In case of nested loops, the control comes out of the loop in which the `break` statement is located and not from all the outer loops. Thus if the `break` is executed within the inner loop, it will come out of the inner loop into the outer loop.

The following example illustrates the use of the `break` statement to find the sum of an arbitrary number of positive values input by the user.

```
1   /*Program-48: Use of break statement*/
2   #include<stdio.h>
3   int main()
4   { int count=0, num, sum=0;
5     printf("\nEnter numbers to add. (Enter a negative number to end list): ");
6     while(1)
7       { scanf("%d", &num);
8         if(num<0)
9             break;
10        sum+=num;
11        ++count;
12      }
13    printf(("\nThe sum of %d numbers is %d", count, sum);
14    return 0;
15  }
```

Output:

```
Enter numbers to add. (Enter a negative number to end list):
4
6
8
-1
The sum of 3 numbers is 18
```

In line-6 we have used an **infinite `while` loop** i.e. a loop whose condition part is always True, to enter the numbers to add. The loop is supposed to end if someone enters a negative value in line-7. If a positive number is entered, the condition in line-8 gets False and the control goes to line-10, skipping line-9 and adds the number to the existing sum. A counter variable `count`, used to keep track of the number of values entered, is incremented in line-11. After this the control goes back to line-6, where the condition `while(1)` being always True, the control again enters the body of the loop. The loop goes on repeating in this way till the user enters a negative number, when the condition in line-8 gets True and hence the `break` statement in line-9 gets executed. It simply takes the program control out of the loop to line-13, skipping the rest of the loop and prints the total `sum` in line-13.

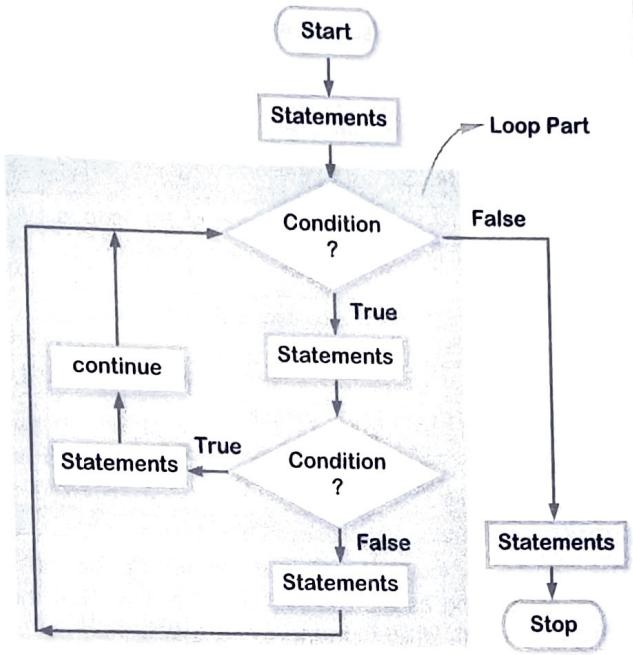
11.5 The continue statement

The `continue` statement is used to **take the control to the beginning of the loop, bypassing certain statements** within the body of the loop. An `if` statement within the body of the loop is used to check some condition. If the condition becomes True, **the program control skips the rest of the statements within** the body of the loop and **goes back to the beginning of the loop.** There it checks the condition again.

The flowchart on the right shows the functioning of the `continue` statement in general.

Whereas the `break` statement goes out of the loop, the `continue` statement continues to execute the loop, **skipping some statements.**

The following program piece illustrates the working of the `continue` statement to print odd numbers from 1 to a number as entered by the user.



The continue statement

```
1  /*Program-49: Example of continue statement*/
2  #include<stdio.h>
3  int main()
4  { int count=1, odd=0, n;
5    printf("\nEnter the number up to which to print odd numbers starting from 1: ");
6    scanf("%d", &n);
7    while( count<=n )
8      { if(count%2==0)
9          {++count;
10          continue;
11         }
12       printf("\n%d", count);
13       ++count;
14       ++odd;
15     }
16   printf("\nTotal number of odd numbers printed is %d", odd);
17   return 0;
18 }
```

In case the *if* condition is True, the program control goes to the condition part of *while* statement again, skipping the remaining part of the loop body

The continue statement is used to skip a certain portion of the loop code and go to the loop header directly.

Output:

```
Enter the number up to which to print odd numbers starting from 1: 8
1
3
5
7
Total number of odd numbers printed is 4
```

The value up to which the odd numbers are to be printed is entered in line-6 in the variable `n`. The counter variable `count` has been initialised to 1 in line-4. It is used to run the loop `n` number of times. The `while` loop condition compares the value of `count` and the value of `n`. As long as the condition is True, the loop executes.

Within the loop in line-8 the `if` condition checks if `count` value is odd or even by using remainder division operation. If the condition becomes True, it implies that the `count` value is even. In that case within the `if` block, in line-9 the `count` value is incremented by 1 and in line-10 the `continue` statement forces the program control to skip the part from line-12 to 14 and again check the condition part in line-7 with the incremented value of `count`.

In case the **if** condition is False, the **continue** statement is not executed and the lines-12 to 14 get executed and print the odd count value. In that case the **count** value is incremented in line-13. Another variable called **odd** is also incremented in line-14. It is used to count the number of odd values printed.

When the loop terminates, the value stored in the odd variable is printed in line-16.

Difference between break and continue

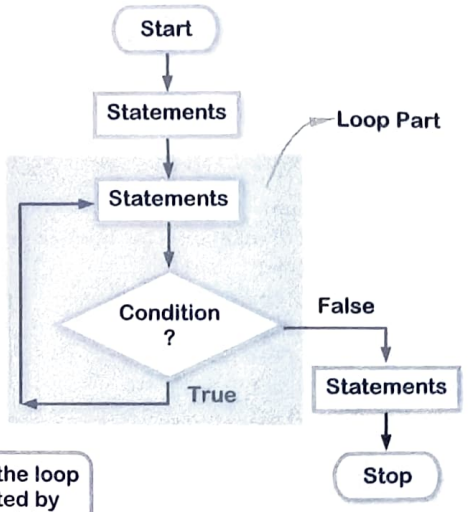
Difference between a break and a continue statement

Break	Continue
A break statement is used to **come out of the loop** in which it is placed in case certain conditions are satisfied.	A continue statement is used to **skip execution of certain portions of the loop** for certain conditions and continue for other conditions.
A break statement takes the **control out of the loop** in which it is placed.	A continue statement takes the **control to the beginning of the loop**, skipping certain portions.
E.g. `while( x++>0 ) { if(x==100) break; printf ("\n%d", x); }`	E.g. `while( x++ <100 ) { if ( x>50 && x<75 ) continue; printf ("\n%d", x);}`

The do-while loop

11.6 The do-while Loop

The **do-while** loop is similar to the **while** loop in that it continues as long as the condition remains True. But the main difference is that, while the **while** loop may or may not execute at all depending upon the condition, the **do-while loop will execute at least once** before testing the condition. The reason is apparent from the flow chart depicting the structure of the **do-while** loop.



The **do-while loop** is an exit controlled loop i.e. it checks the condition at the end of the loop body

The **condition** in a **do-while** loop is **tested at the end of the do-while block** and hence the block will execute at least once before testing the condition. It is thus an **exit controlled** loop structure. The syntax of the **do-while** loop is:

```
initialise loop counter; /*optional*/
do { statements;
     … … … …;
     increment loop counter; /*optional*/
   } while(condition);
```

Body of the loop indicated by enclosing them within the curly braces { }

Semicolon after the while condition

The following program inputs a temperature in the Kelvin scale and converts it to Celsius scale. The input is tested for validity (i.e. greater than or equal to 0) using a **do-while** loop.

Checking for a valid input

```
1   /*Program-50: To enter and check for a valid input using do-while loop*/
2   #include<stdio.h>
3   int main()
4   {float Ktemp, Ctemp;
5    int check=1;
6    do { if(check==1)
7              puts("\nEnter value of temperature in the Kelvin scale: ");
8        else
9              puts("\nPlease input a valid temperature!");
10       scanf("%f", &Ktemp);
11       check=0;
12     } while (Ktemp < 0);
13  Ctemp = Ktemp - 273;
```

```
14  printf("\n%f Kelvin = %f degree Celsius", Ktemp, Ctemp);
15  return 0;
16 }
```

The variable **Ktemp** is used to input the temperature in Kelvin scale, which has to be a positive value. The integer variable **check** is used to check whether the **do-while** loop has been executed once or not and is initialised to 1 in line-5. The **do-while** loop starts in line-6. After entering the loop body the program encounters the **if** statement in line-6. As the value of **check** has been initialised to 1, the condition of the **if** statement becomes True in line-6 and the program prints **Enter value of temperature in the Kelvin scale:**. Next the program goes to line-10, skipping the **else** statement and inputs the temperature using the **scanf()** function. In line-11 the variable **check** is made equal to 0.

The code in a do-while loop will run at least once

Suppose the user enters a positive value as required. Then **Ktemp > 0**, and hence the condition in line-12 becomes False and the program control goes out of the loop to calculate **Ctemp** in line-13.

In case someone enters a negative value then **Ktemp<0** and the **while** condition in line-12 becomes True. Thus the program control goes back to the start of the loop i.e. to line-6. However now the **if** condition becomes false as the variable **check** had been re-initialised to 0 in line-11. Thus the **else** statement gets executed and the program prints **Please input a valid temperature!** i.e. a different prompt. The new value of **Ktemp** is again input in line-10 and the condition tested in line-12. As long as the user inputs a negative value, the condition will be satisfied and the loop will go on repeating. Whenever the user inputs a positive value the **do-while** condition becomes False and the loop terminates.

A do-while loop can be used to correctly input a value within a given range of values.

Therefore to test for a valid input the condition of the do-while loop should be just the reverse of the required or valid input.



Test for a valid input using do-while loop

In case one does not use different prompts for an invalid input, then the **loop can be simplified as shown below, without using the check** variable. In this case if someone enters an invalid input then the condition becomes True but the program prints the same prompt "**Enter value of temperature in the Kelvin scale:**" as it had printed in the first run.

```
1  do { puts("\nEnter value of temperature in the Kelvin scale: ");
2       scanf("%f", &Ktemp);
3     } while (Ktemp < 0);
```

We had said that the **do-while** loop is most suited for cases where **the loop needs to run at least once**. One such case is to find the **HCF (or GCD) of two numbers** input by the user as discussed below.



HCF or GCD of two numbers

```
1  /*Program-51: To find the HCF or GCD of two numbers using do-while loop*/
2  #include<stdio.h>
3  int main()
4  { int divisor, dividend, rem, hcf;
5    Printf("\nEnter number1: ");
6    scanf("%d", &dividend);
7    Printf("\nEnter number2: ");
8    scanf("%d", &divisor);
9    do{ rem = dividend % divisor;
10       dividend=divisor;
11       divisor=rem;
12     } while(rem != 0);
13   hcf = dividend;
14   Printf("\nThe required HCF = %d", hcf);
15   return 0;
16 }
```

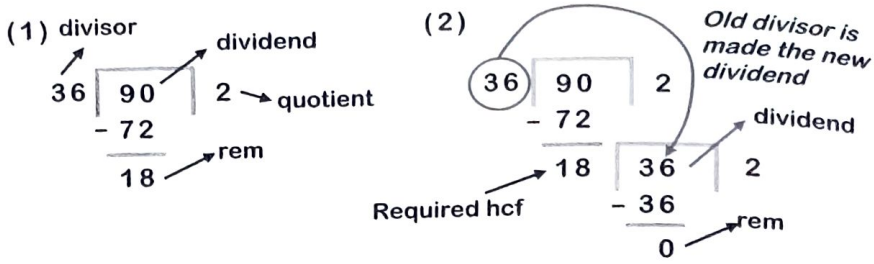
Output:

```
Enter number1: 90
Enter number2: 36
The required HCF = 18
```

Logic behind finding HCF or GCD

The **algorithm** used for calculating the `hcf` of two numbers is shown on the right for numbers 90 and 36.

First the `dividend` (i.e. 90) is divided by the `divisor` (i.e. 36) to get a quotient of 2 and a remainder `rem` of 18 (It **does not matter which is greater**).



The process of **division is repeated till we get a `0` remainder**. In doing so, in step (2) the `divisor` of the last division is made the new `dividend` and the remainder `rem` from the last division is made the new `divisor` and the division performed. This time we get a `0` **remainder** and hence we **stop the division**. The required `hcf` is the current `divisor` i.e. 18 in this case (for which `rem=0`).

The algorithm shown above is followed in writing the above code. After entering the `dividend` and the `divisor` in line-6 and 8, the program enters the `do-while` loop. Within the loop the remainder `rem` is obtained by using the remainder operator in line-9. Next the new `dividend` is obtained in line-10 by assigning it to the old `divisor`. Similarly the **new `divisor` is made equal to the old remainder `rem`** in line-11. Finally the condition is tested in line-12. As we had said, **the process of division would continue till we get a 0 remainder**. Hence if `rem!=0`, the condition becomes True and the loop is repeated as in the first step of our example with the numbers 36 and 90 and a remainder of 18.

When the loop is repeated for the second time, the value of remainder `rem` becomes 0 in line-9 for the `divisor=18` and the `dividend=36`. Next in line-10, the new `dividend` is made equal to the old `divisor` i.e. 18. **Hence now the `dividend` contains the value 18**. Next in line-11 the `divisor` is assigned the value of `rem` i.e. 0 in this case. When the condition is checked in the next line i.e. line-12, it is found to be False as `rem=0` now and the **loop terminates**.

From the algorithm we find that the required `hcf` is 18 as we got a remainder of 0 in this case. This `divisor` value is however stored in the current `dividend` vide line-10. Hence the value of the `dividend` is now assigned to the variable `hcf` in line-13 and printed in line-14.

Difference between while and do-while loop

Difference between while and do-while

While	Do-While
In a while loop the **condition is checked first** and then the body of the loop is executed.	In a do-while loop the body of the loop is executed first and **the condition is checked at the end**.
The body of a while loop **may or may not be executed**, depending upon the condition.	The body of a do-while loop **is executed at least once** as the condition is checked at the end.
Structure: while (condition) { statements; }	**Structure:** do { statements; } while (condition);
There is **no semicolon** after the `while` condition.	There is a **semicolon** after the `while` condition.
e.g. `while( x++<100 ) printf("\n %d", x);`	e.g. `do{puts("Enter +ve no."); scanf("%d",&num); } while (num < 0);`

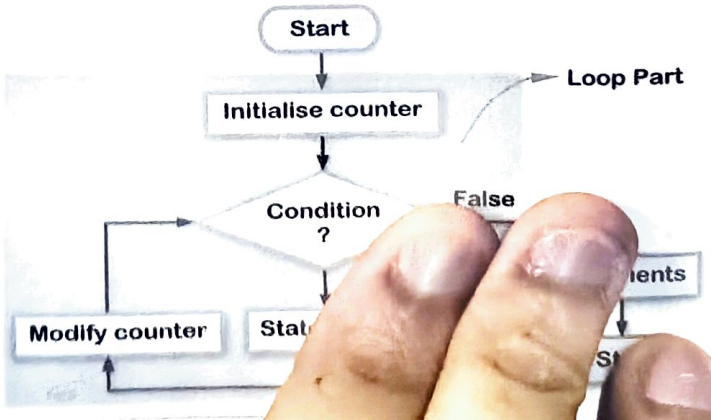
The for loop

11.7 The for Loop

The **most important** and **widely used** of the loop structures is the `for` loop. It can be used in place of the `while` or `do-while` loops with proper modification.

The **flowchart** on the right shows the functioning of the `for` statement. The counter initialisation, condition testing and counter modification – all can be done within the header portion of the `for` loop.

The `for` loop is normally used to run a loop a fixed number of times.



Body of loop indicated by enclosing them within the curly braces { }

```
for (initialise_loop_counter; check_condition; modify_counter)
   { statements;
     ... ... ... ...;
   }
```

Semicolon after each part and NOT a comma

The `for` statement executes in the following **stepwise manner**:

1. The **loop counter** is first **set** to an initial value
2. Next the loop **condition is tested**
3. If the **condition** is found **True**, the **body of loop is executed**
4. Next the **loop counter** is incremented/decremented i.e. **modified**
5. The **condition is again tested** and if found True then body of loop is again executed
6. The process is **repeated** till the **condition becomes False**

Working of *for* loop

Let us now repeat the problem of finding the factorial of a number, but now by using a `for` loop.

Factorial using *for* loop

```
1  /*Program-52: To find the factorial of a number using a for loop*/
2  #include<stdio.h>
3  int main()
4  { int num, i;
5    long int fact=1;
6    printf("\nEnter number to find factorial: ");
7    scanf("%d", &num);
8    for( i=1; i<=num; i++ )
9       fact = fact * i;
10   printf("\nThe factorial of %d is %ld", num, fact);
11   return 0;
12 }
```

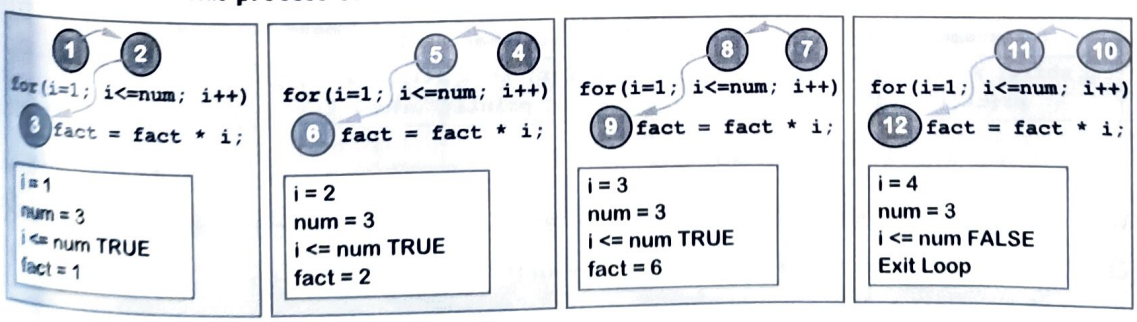
Output:

```
Enter number to find factorial: 3
The factorial of 3 is 6
```

The functioning of the above program is shown by the diagram below. The `for` loop starts in line-8. Since the body of the loop has a single statement, we have not used the curly braces to enclose the body of the loop. The counter generally used in a `for` loop is taken as `i`.

The numbering below shows the sequence in which the program is executed. The variable `i` is initialized to `1` at the beginning of the `for` loop in line-8. Next the condition is checked. If the condition is found to be True the body of the loop gets executed in line-9. Next the loop counter variable `i` is incremented by the statement `i++` and the condition is again tested. If the condition is found to be True then the body of the loop gets executed. This **process continues till the condition becomes false when the loop is ended.**

Each of the arguments of the for loop header is optional and can be selectively used.



The **three sections in the `for` header are optional** and may be selectively omitted if not required. However, the two **semicolons** must be kept. Thus the following piece of code is perfectly correct:

Different formats of *for* loop

```
1  int i=1;                /*Loop counter i is initialised to 1 outside the loop*/
2  for( ;i<=10; )          /*Loop header contains only the condition part*/
3     {printf("\n%d", i);  /*Loop body executes loop code*/
4      ++i;}               /*Loop counter incremented at the end of the loop body*/
```

The next program finds the **sum of the terms of an A.P. series** up to a certain number of terms.

```
1   /*Program-53: To find the sum of an AP series using a for loop*/
2   #include<stdio.h>
3   int main()
4   { float sum=0, term, ft, cd;
5     int n, i;
6     printf("\nEnter first term of A.P.: ");          scanf("%f", &ft);
7     printf("\nEnter common difference of A.P.: ");  scanf("%f", &cd);
8     printf("\nEnter number of terms to add: ");     scanf("%d", &n);
9     term = ft;
10    for( i=1; i<=n; i++ )
11       { sum += term;       /*Same as sum = sum + term i.e. series sum relation */
12         term += cd;        /*Same as term = term + cd */
13       }
14    printf("\nThe required sum up to %d terms is %0.2f", n, sum);
15    return 0;
16  }
```

Output:

```
Enter first term of A.P.: 1.5
Enter common difference of A.P.: 2.5
Enter number of terms to add: 10
The required sum up to 10 terms is 127.50
```

In the above program note that the variable **`term` is initialised with the value of the first term** `ft` in line-9. This serves as the value of the first term that is added to the variable `sum` in line-11. The **new term is then calculated** in line-12 by adding the current `term` with the common difference `cd`. The loop counter is then incremented by `i++` and the condition tested again. If found True, the new `sum` is calculated in line-11 by adding the newly generated `term` to the previous `sum`. This continues till the condition becomes False and the loop is terminated. The final `sum` up to n terms is printed in line-14.

Difference between while and for loops

Difference between a while and a for loop

While	For
The **while** statement is **usually used** to perform an operation for an **unknown number of times.**	The **for** loop is **usually used** to perform an operation a **fixed number of times.**
The condition portion of the 'while' statement contains **only the condition** to be satisfied.	The condition portion of the 'for' statement can contain the **initialisation, condition and counter increment.**

Structure:
`while( condition )`
`{ statements; }`
	Structure:
	`for(initialise_counter; condition; modify_counter)`
	`{ statements; }`

E.g. `while( x++<100 )` `printf("\n%d", x);`	E.g. `for( x=0; x<100; x++ )` `printf("\n%d", x);`

Using nested for loops

- Using nested *for* loops:

We had earlier seen the nesting of `while` loops. The next example demonstrates the use of nested `for` loop.

Calculate the value of the series e^x given by: $e^x = 1 + x/(1!) + x^2/(2!) + x^3/(3!) + x^4/(4!) + \ldots$ to n terms.

In a nested loop, the inner loops runs fully for each run of the outer loop.

```
1   /*Program-54: To find the value of e^x using a nested for loop*/
2   #include<stdio.h>
3   int main()
4   { int n, i, j, k;;
5     float e = 1.0, x, power, factorial; /*As starting value, e is initialised to 1*/
6     printf("\nInput the value of x: "); scanf("%f", &x);
7     printf("\nInput number of terms to include: ");
```

```
8    scanf("%d", &n);
9    for(i=1; i<n; i++)                  /*Outer for loop to count number of terms added*/
10      { power = 1; factorial = 1;
11        for(j=1; j<=i; j++)            /*Nested inner for loop to calculate the power*/
12           power = power*x;
13        for(k=1; k<=i; k++)            /*Nested inner for loop to calculate the factorial*/
14           factorial = factorial*k;
15        e += power/factorial;          /*new term added to existing value of e*/
16      }
17   printf("\nThe value of e^x up to %d terms is %f", n, e);
18   return 0;
19 }
```

Output1:

```
Input the value of x: 1
Input number of terms to include: 10
The value of e^x up to 10 terms is 2.718282
```

Output2:

```
Input the value of x: 2
Input number of terms to include: 4
The value of e^x up to 4 terms is 6.333333
```

The nested loops used in the program are highlighted below:

```
9    for(i=1; i<n; i++)
10      { power = 1; factorial = 1;
11        for(j=1; j<=i; j++)
12           power = power*x;
13        for(k=1; k<=i; k++)
14           factorial = factorial*k;
15        e += power/factorial;
16      }
```

Outer *for* loop

Inner *for* loop-1 to calculate the *power*

Inner *for* loop-2 to calculate the *factorial*

Single statement within Inner for loop

Let us now examine the functioning of the above program section by taking the number of terms to calculate the sum as 4 i.e. `n=4` and the value of `x=2`.

Run	i	j	factorial	k	power	e = e + power/factorial
1	1	1	1=1!	1	1*2=2	1 + 2/1! = 3
2	2	1	1*2=2!	1	1*2*2=4	3 + 4/2! = 5
		2		2		
3	3	1	1*2*3=3!	1	1*2*2*2=8	5 + 8/3! = 6.333333
		2		2		
		3		3		

The first term of the series i.e. 1 is being taken care of in the initial value of `e` in line-5. We find that **excepting the first term**, all other terms of the series follow a certain pattern. If we **take the second term of the series** i.e. x/1! as **term number 1** then the rth term of the series can be written as: $\mathbf{x^r / (r)!}$

Accordingly the computations of the next terms are being carried by the subsequent `for` loops following this pattern. Since `e` has been initialised to 1 (first term), the number of terms counted by `i` goes up to `(n-1)` in the outer `for` loop hence the condition is `i<n` and not `i<=n` in line-9.

At each run of the `for` loop new values of the variables `power` and `factorial` (taken as a `float` to accommodate large powers & factorials) are calculated based on the term number. For example when `i=1` i.e. for term number 1, $x^1/1!$ is calculated. For `i=2`, i.e. for the second term $x^2/2!$ is calculated. For `i=3`, $x^3/3!$ is calculated and so on. After the calculation of a new term, it is then added to the existing sum stored in the variable `e` to get the new value of `e`. Let us analyse the program for `n=4`.

1. In line-9 the outer `for` loop starts. This loop is used to count the number of terms to add.
2. In line-10 the variables `power` and `factorial` are initialised to `1`.
3. Line-10 and 11 form the first nested `for` loop used to calculate the power of `x` for that term. When `i=1` the `j` **counter will run only once** and `power = power*x` statement in the `for` loop calculates x^1.
4. Line-13 and 14 form the second nested `for` loop that is used to calculate the factorial for that term. When `i=1` the `k` **counter will run only once**, and hence the `factorial = factorial*k` statement in the `for` loop calculates the factorial as `1`.
5. Line-15 then adds the new value of `power/factorial` to the existing value of `e`.
6. The program control then goes back to line-9 where the counter `i` is incremented to `2` and the code from step2 above is again repeated. This time since `i=2`, the `j for` **loop will run twice** and calculate x^2. Similarly the `k for` **loop will also run twice** and calculates the value of `2!`. Line-15 then adds the current value of `power/factorial` to `e`.
7. The program control again goes back to line-9 where the counter `i` is incremented to `3` and the code from step2 above is again repeated. This time since `i=3`, the `j for` **loop will run thrice** and calculate x^3. Similarly the `k for` **loop will also run thrice** and calculates the value of `3!`. Line-15 then adds the current value of `power/factorial` to `e`.
8. After this, when `i` is incremented to `4` in line-9, the condition becomes False and the loop is ended. Line-17 finally prints the final value of e^x.

Go to page P1-11-35 for a more efficient code to find e^x.



Using comma operator

Use of the Comma operator:

The comma operator in general **allows two expressions to appear at a place**, where otherwise only one expression would have been used. The general syntax of the comma operator is:

```
for(expression_1a, expression_1b, ...; expression_2; expression_3a, expression_3b, ...)
```

Multiple initialisations | Single Condition | Multiple modifications

The comma operator is used to write two expressions at a place separated by commas

The expressions **expression_1a, expression_1b,** etc. separated by commas are used to initialise different variables simultaneously within the loop header. Similarly expressions **expression_3a, expression_3b,** etc. separated by commas are used simultaneously within the loop header to **modify the variables**. Though multiple statements can be used in the initialisation and in the modification portions, but usually **only one test expression** or condition is used, as given by **expression_2**. The single test expression can however have conditions **connected by logical operators**. The following program shows use of comma operator.



Finding sin-x using series

Calculate the **value of sin(x)** given by **sin(x)** = $x/(1!) - (x^3)/(3!) + (x^5)/(5!) - (x^7)/(7!) + (x^9)/(9!)$...

```
1   /*Program-55: To find the value of sin(x) using a nested for loop*/
2   #include<stdio.h>
3   int main()
4   { int n, i, j, k, sign;
5     float sin = 0.0, x, power, factorial;
6     printf("\nInput the value of the angle x (in radians): ");
7     scanf("%f", &x);
8     printf("\nInput number of terms to include: ");
9     scanf("%d", &n);
10    for(i=1, sign=1; i<=n; i++, sign *= -1)
11       { power = 1; factorial = 1;
12         for(j=1; j<=2*i-1; j++)
13            power = power*x;
14         for(k=1; k<=2*i-1; k++)
15            factorial = factorial*k;
16         sin += sign*power/factorial;
17       }
```

```
18    printf("\nThe value of sin(%f) up to %d terms is %f", x, n, sin);
19    return 0;
20 }
```

Output:

```
Input the value of the angle x (in radians): 0.523598
Input number of terms to include: 10
The value of sin(0.523598) up to 10 terms is 0.499999
```

The above program is **similar to the previous program** i.e. program-54. Note the use of comma operator in line-10 to initialise variables `i` and `sign` to `1` and to modify them again using the `i++` and `sign*=-1` statements. Also note that the condition `i<=n` is used to count the number of terms already added.

From the series we find that the **power of** `x` and the **factorial value** for that term **are the same**. Moreover the **power forms an A.P. series** 1, 3, 5, 7 etc. whose i^{th} term is given by `(1+(i-1)*2) = 2*i-1`. Thus the `j` counter in the `for` loop in line-12 and the `k` counter in the `for` loop for line-14 go up to `2*i-1`. For example if `i=3`, i.e. for the **third term**, `j` and `k` will go up to `(2*3-1)=5` i.e. will calculate x^5 and `5!`.

Apart from the term values, the **sign of the individual terms also alternate** starting from a positive sign. For this we have taken a variable called `sign` which is initialised to `1` in line-10. At the end of execution of the outer loop, `sign` is modified by multiplying it by `(-1)` using the `sign*=-1` statement in line-10. Thus if `sign=1`, it will get converted to `(-1)` and if it is `(-1)`, it will get converted to `1`. This `sign` value is multiplied with the term and added to the existing value of `sin` in line-16. Remember **x is in radians**.

Go to page P1-11-36 for a more efficient code to find sine(x).

The following program piece shows what happens when the **comma operator is used in the test-condition** checking part of a loop.



Comma operator used with a condition

```
1  /*Program-56: Use of comma operator in the test condition*/
2  #include<stdio.h>
3  int main()
4  { int i, j, m=7, n=5 ;
5    for( i=1, j=1; i<=m, j<=n ; i++, j++ )
6        printf("\n i=%d, j=%d", i, j);
7    return 0;
8  }
```

Output:

```
i=1, j=1
i=2, j=2
i=3, j=3
i=4, j=4
i=5, j=5
```

From the output we find that the loop runs based on the condition `j<=n`, the second condition, with the value of `n=5`. The first condition `i<=m` is ignored. If we reverse the position of the conditions in line-5 as shown in the following code, then the output becomes:

When two conditions are separated by commas, then the second condition is executed.

```
5    for( i=1, j=1; j<=n, i<=m ; i++, j++ )
6        printf("\n i=%d, j=%d", i, j);
```

Output:

```
i=1, j=1
i=2, j=2
i=3, j=3
i=4, j=4
i=5, j=5
i=6, j=6
i=7, j=7
```

We find that again the second condition `i<=m` is tested and the first condition is ignored as in the last case.

11.8 Some worked out examples

Reverse a number input by the user and check if it is a palindrome.



Checking for a Palindrome number

```
1   /*Program-57: Reversing a number and checking if it is a palindrome*/
2   #include<stdio.h>
3   int main()
4   { long int num, copy, sum=0;
5     int digit;
6     printf("\nEnter number to reverse and check: ");
7     scanf("%ld", &num);
8     copy = num;
9     while( num > 0 )
10      {digit = num%10;
11       sum = sum*10 + digit;
12       num = num/10;
13      }
14    printf("\nThe reversed number is %ld", sum);
15    if (copy == sum)
16      printf("\nNumber is a palindrome");
16    else
18      printf("\nNumber is not a palindrome");
19    return 0;
20  }
```

Always keep a copy of the original number if you want to compare the original number later

Output1:

```
Enter number to reverse and check: 152
The reversed number is 152
Number is not a palindrome
```

Output2:

```
Enter number to reverse and check: 25752
The reversed number is 25752
Number is a palindrome
```

To reverse a number we simply take out each digit from the right of the number one after the other **using the remainder operator** and follow the procedure as indicated for the number 152:

- In line-7 the value `152` is entered into the variable **`num`**. The same is copied to **`copy`** in line-8
- In line-9, as `152 > 0`, the **`while`** condition gets satisfied and the program enters the loop
- In line-10 **`digit = num%10 = 152%10 = 2`** is obtained (remainder of `152` divided by `10`)
- In line-11 **`sum = sum*10+digit`** gives **`sum = 0*10+2 = 2`** (as **`sum`** has been initialised to `0` in line-4)
- In line-12, the new value of **`num = num/10 = 152/10 = 15`** (result of integer division of `152` by `10`)
- Control goes back to line-9, where condition is True as new value of **`num=15 > 0`**.
- In line-10 **`digit = num%10 = 15%10 = 5`** is obtained (remainder of `15` divided by `10`)
- In line-11 **`sum = sum*10+digit`** gives **`sum = 2*10+5 = 25`** (as previous value of **`sum=2`**)
- In line-12, the new value of **`num = num/10 = 15/10 = 1`** (result of integer division of `15` by `10`)
- Control again goes back to line-9, where condition is True as new value of **`num=1`** is **`> 0`**.
- In line-10 **`digit = num%10 = 1%10 = 1`** is obtained (remainder of `1` divided by `10`)
- In line-11 **`sum = sum*10+digit`** gives **`sum = 25*10+1 = 251`** (as previous value of **`sum=25`**)
- In line-12, the new value of **`num = num/10 = 1/10 = 0`** (result of integer division of `1` by `10`)
- Control again goes back to line-9, where the condition becomes False as new value of **`num=0`**.

Thus the value of the variable **sum** obtained after the loop has ended is equal to **251** which is the reverse of **152**. The **if** condition in line-15 checks if the original value of **num** as stored in the variable **copy** is equal to **sum** or not. In case of a palindrome the reverse and the original value would be the same (as in case of the number 25752). In this case **251** and **152** are not identical and hence the **if** condition becomes False and the **else** statement prints "**Number is not a palindrome**".

Note that you have to keep a **copy** of the original number to compare it with the calculated **sum** later. This is because the **while** loop statement **num=num/10** will reduce the original value of **num** to **0** when the loop ends. Hence the result of the comparison with the original number will always be False. If a copy of the original number is kept, then the copied number can be compared with the **sum** to check the condition later.

Program to check if a number is such that the sum of the factorial of digits of the number is equal to the number (also called a Facterone). For example 145 = 1! + 4! + 5!



Checking for a Facterone

```
1  /*Program-58: Check if a number is a Facterone*/
2  #include<stdio.h>
3  int main()
4  { long int num, copy, fact, sum=0;
5    int digit, count;
6    printf("\nEnter number to check if it is a Facterone: ");
7    scanf("%ld", &num);
8    copy = num;
9    while( num > 0 )
10     {digit = num%10;
11      fact=1;
12      count=1;
13      while( count <= digit )
14            {fact = fact*count;
15             count++;
16            }
17      sum = sum + fact;
18      num = num/10;
19     }
20   if (copy == sum)
21     printf("\nNumber is a Facterone");
22   else
23     printf("\nNumber is not a Facterone");
24   return 0;
25 }
```

Output1:

```
Enter number to check if it is a Facterone: 145
Number is a Facterone
```

Output2:

```
Enter number to check if it is a Facterone: 214
Number is not a Facterone
```

The digits from the input number are extracted using the outer **while** loop of line-9 using the code **digit=num%10**. The inner **while** loop of line-13 then calculates the factorial of the extracted digit. The factorial is then added to the **sum** in line-17.

Note than the variables **fact** and **count** are initialised in line-11 and 12 respectively within the outer loop. This is because every time a new factorial is calculated with a given digit and added to the sum, the variables **fact** and **count** need to be re-initialised to 1 to use them for calculating the factorial of the next digit. If these are initialised outside the outer loop, then they will retain their value after calculating the factorial of the first digit and will give a wrong result when the factorials of the subsequent digits are calculated.

The following program uses the **for** loop to **print a conversion table from Centigrade to Fahrenheit**, the starting and ending temperatures being input by the user.

```
1   /*Program-59: Centigrade to Fahrenheit conversion Table*/
2   #include<stdio.h>
3   int main()
4   { int i, cen_start, cen_stop; float Fahrenheit;
5     printf("\nEnter the temp. in Centigrade to start:");
6     scanf("%d", &cen_start);
7     printf("\nEnter the temp. in Centigrade to stop:");
8     scanf("%d", &cen_stop);
9     for(i=cen_start; i<=cen_stop; i++)
10       { Fahrenheit = (9.0/5.0)*i+32;
11         printf("\n%d deg. C = %.2f deg. F", i, Fahrenheit);
12       }
13    return 0;
14  }
```

Output:

```
Enter the temp. in Centigrade to start:20
Enter the temp. in Centigrade to stop:30
20 deg. C = 68.00 deg. F
21 deg. C = 69.80 deg. F
22 deg. C = 71.60 deg. F
23 deg. C = 73.40 deg. F
24 deg. C = 75.20 deg. F
25 deg. C = 77.00 deg. F
26 deg. C = 78.80 deg. F
27 deg. C = 80.60 deg. F
28 deg. C = 82.40 deg. F
29 deg. C = 84.20 deg. F
30 deg. C = 86.00 deg. F
```

In the above example both the **counter initialisation** value and the **condition checking** value have been **input by the user** and are variable terms instead of fixed ones. Therefore `i` is not initialised to 1 as before but to `cen_start` and the loop runs till the value of `i` becomes equal to `cen_stop`.

In the above example we have incremented the loop counter at each run of the loop by 1. In general the **counter can be incremented by any value** as shown in the example below **to print all multiples of 3**.

```
1   /*Program-60: To print the multiples of 3*/
2   #include<stdio.h>
3   int main()
4   { int i, limit;
5     printf("\nEnter the value up to which to print all multiples of 3: ");
6     scanf("%d", &limit);
7     for(i=3; i<=limit; i+=3)
8         printf("%d, ", i);
9     return 0;
10  }
```

Output:

```
Enter the value up to which to print all multiples of 3: 25
3, 6, 9, 12, 15, 18, 21, 24,
```

Since we want to print the multiples of 3, we have initialised loop counter `i` to the starting multiple i.e. 3. After checking the condition and printing the first multiple, the loop counter `i` is incremented by 3 using the code `i+=3` i.e. `i=i+3` in line-7. By this we get the next multiple i.e. 3+3=6, which is then printed in line-8. This continues till `i` reaches the limiting value given by the variable `limit`.

To find whether a number entered by the user is a prime number or not:



Checking for a prime

```
1  /*Program-61: To check whether a number is prime*/
2  #include<stdio.h>
3  int main()
4  { int num, i, check=1;
5    printf("\nInput number to find if it is prime: ");
6    scanf("%d", &num);
7    for(i=2; i<num; i++)
8       { if(num%i == 0)
9           { check=0;
10            break;
11          }
12       }
13   if(check==1)
14       printf("\nThe number %d is prime", num);
15   else
16       printf("\nThe number %d is not prime", num);
17   return 0;
18 }
```

Output1:

```
Input number to find if it is prime: 5
The number 5 is prime
```

Output2:

```
Input number to find if it is prime: 27
The number 27 is not prime
```

A prime number is a number which is **divisible by 1 and itself only**. Thus to check if a number is prime, we simply divide the number by all numbers greater than 1 and less than the number. In case it is not divisible by any of these numbers then it is prime. Thus **to check if 5 is a prime number we divide it by 2, 3, and 4** (1 and 5 are excluded as per the definition of a prime number) and hence the `for` loop runs from `i=2` to `i<num` i.e. up to `4` and **NOT** up to `i<=num` i.e. `5`.

The `for` loop starts from line-7 and initialises the loop counter `i` to `2`. If the entered value `num=5`, then the condition `i<num` becomes True and the control enters the loop. In line-8, the `if` statement checks if `num%i` i.e. the remainder of dividing `num` by `i` is `0` or not. **If a number is prime, proper division is not possible and the remainder will always be some non-zero value**. Hence the condition in line-8 will always be False for a prime number. Therefore it will never enter the `if` block and will not change the value of the variable `check` to `0` in line-9. In that case it will also never execute the `break` statement.

In case of a non-prime number, for example `27`, the condition `num%i==0` becomes True for `i=3`. Hence the program control enters the `if` block and changes the value of the variable `check` to `0` in line-9. If a number becomes **divisible by any number then it is not prime** and hence it is not required to check the number any further. The loop is terminated forcibly in that case by using the `break` statement in line-10. Once the `break` statement in line-10 breaks out of the loop it takes the program control straight to line-13 skipping the rest of the loop, as shown in the next page.

```
7     for(i=2; i<num; i++)
8        { if(num%i == 0)
9            { check=0;
10             break;
11           }
12       }
13   if(check==1)
```

In line-13, the `if` statement simply checks the value of the variable `check`. If `check==1`, i.e. the original value, the program had not entered the `if` block and hence prints that the number is prime. Else if

`check==0` then the program had entered the `if` block where it got modified to 0 and hence prints that the number is not prime.

The series 1, 1, 2, 3, 5, 8, 13, 21... is known as the Fibonacci series. The speciality of the series is that except for the first two terms, all other terms are derived by adding the previous two terms. The first two terms are fixed terms and have the value 1. For example the 6th term i.e. 8 is obtained by adding the previous two terms i.e. 3 and 5 as 3+5=8.



Printing Fibonacci series

```
1   /*Program-62: To print the Fibonacci Series up to n terms using a for loop*/
2   #include<stdio.h>
3   int main()
4   { int i, n;
5     long int FirstTerm = 1, SecondTerm = 1, CurrentTerm;
6     printf("\nEnter the number of terms to print: ");
7     scanf("%d", &n);
8     if(n==1)
9        printf("\nTerm-1 is = 1");
10    if(n>=2)
11      {printf("\nTerm-1 is = 1");
12       printf("\nTerm-2 is = 1");
13      }
14    for(i=3; i<=n; i++)
15      { CurrentTerm = FirstTerm + SecondTerm;
16        printf("\nTerm-%d is = %ld", i, CurrentTerm);
17        FirstTerm = SecondTerm;
18        SecondTerm = CurrentTerm;
19      }
20    return 0;
21  }
```

Output1:

```
Enter the number of terms to print: 2
Term-1 is = 1
Term-2 is = 1
```

Output2:

```
Enter the number of terms to print: 6
Term-1 is = 1
Term-2 is = 1
Term-3 is = 2
Term-4 is = 3
Term-5 is = 5
Term-6 is = 8
```

The first 2 terms of the Fibonacci series being the same, are printed outside the loop. The next terms are printed using the `for` loop which generate each successive term using the algorithm:

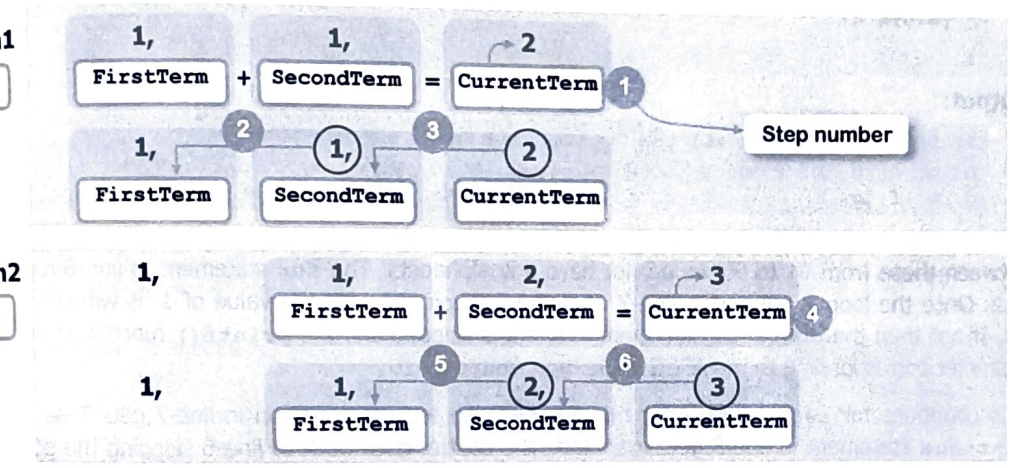
current term = (first term) + (second term)

Since the first 2 terms have already being printed, the loop counter `i` has been initialised to `3` in line-14 and is used to count the remaining terms printed. After printing each new term, the counter variable `i` is increased by `i++` in line-14. This process continues till `i` becomes equal to `n`, when the last i.e. nth term is printed. Then `i` increases to n+1, which makes the test condition `i<=n` False and the loop terminates.

Note the following:

- If `n=1` is entered in line-7, then the **condition in line-8 gets true** and the program executes line-9 to print the 1st term of the series i.e. 1. The other conditions in line-10 and line-14 get false and hence no further term is printed.

- If n=2 is entered then the **condition in line-10 gets true** and the first two terms are printed by executing the `if` block from line-11 to line-12. The other conditions in line-8 and line-14 get false and hence no further term is printed.
- If n>2 is entered then the **conditions in line-10 and 14 both get true**. The `if` block from line-11 to line-12 prints the first 2 fixed terms i.e. 1 and 1. The condition of the `for` loop then gets satisfied and the `for` loop block from line-15 to 18 gets executed and prints the remaining terms.
- At each run of the loop, **line-15 is used to calculate the current term** by adding the first two terms. Line-16 then prints the current term.
- Line-17 and line-18 are used to generate the new set of first and second terms for getting the current term in the next run of the loop. The process is shown by the diagram below for two successive runs of the loop with `i=3` and `i=4`.



Working of Fibonacci series program

An alternate form of the Fibonacci program is written below with the use of the **continue** statement.

```
3   int main()
4   { int i, n;
5     long int FirstTerm = 1, SecondTerm = 1, CurrentTerm;
6     printf("\nEnter the number of terms to print: ");
7     scanf("%d", &n);
8     for(i=1; i<=n; i++)
9       { If (i==1 || i==2)
10            { printf("\nTerm-%d is = 1", i);
11              continue;
12            }
13        CurrentTerm = FirstTerm + SecondTerm;
14        Printf("\nTerm-%d is = %ld", i, CurrentTerm);
15        FirstTerm = SecondTerm;
16        SecondTerm = CurrentTerm;
17      }
18    return 0;
21  }
```

Instead of checking individually the conditions for `n=1` and `n=2`, the whole series is printed using the `for` loop of line-8 in the above program. The `if` statement in line-9 checks if the value of `i=1` or `i=2`. If so, the first two terms having the fixed value 1 get printed. The `continue` statement in line-11 then skips the remaining part of the loop code which calculates a new Fibonacci term.

For values of `i>=2`, the `printf()` statement in line-10 prints the first two terms. When the value `i` becomes more than 2, the `if` condition in line-9 becomes False and the part from line-13 to line-16 that calculates a new Fibonacci term from the last two terms get executed.

The following example prints all the capital and small alphabets.

```
1   /*Program-63: To print all the alphabets using continue statement*/
2   #include<stdio.h>
3   int main()
4   { int i;
5     puts("## Program to print all the Capital and Small alphabets ##\n");
6     for(i=65; i<=122; i++)
7        { if( i>=91 && i<=96 )
8             continue;
9          printf("%c, ", i);
10       }
11    return 0;
12  }
```

Output:

```
## Program to print all the Capital and Small alphabets ##
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, a, b, c,
d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z,
```

The ASCII values of capital alphabets are from 65 to 90 and that of small alphabets are from 97 to 122. In between these from 91 to 96 we do not have any alphabets. The **for** statement in line-6 runs for **i=65** to **122**. Once the loop is entered in line-7, the **if** condition checks if the value of **i** is within the range **91** to **96**. If not then the **continue** statement in line-8 is skipped and the **printf()** function in line-9 prints the character corresponding to the ASCII value represented by **i**.

This continues for **i=65** to **90**. After this, for **i=91** to **96**, the condition in line-7 gets True and hence the **continue** statement in line-8 gets executed. The control goes back to line-6 skipping the portion after the **continue** statement and the counter **i** gets incremented by **i++**. The loop again gets executed and till **i<=96**, the **continue** statement skips the **printf()** function in line-9. After this, when **i>96**, the condition in line-7 again becomes false and the program continues to print the lower case letters.



Printing Patters

The next example prints the pattern shown on the right for number of lines=5:

```
    *
   **
  ***
 ****
*****
```

```
1   /*Program-64: Using nested for loops to print pattern-1*/
2   #include<stdio.h>
3   int main()
4   { int n, i, j, k;
5     printf("\nEnter the number of lines to print: ");
6     scanf("%d", &n);
7     for( i=1; i<=n; i++ )          /*Outer loop counts number of lines*/
8      { for(j=1; j<=(n-i); j++)     /*Loop for the blanks in front of the stars*/
9           printf(" ");             /*Prints a blank space*/
10      for(k=1; k<=i; k++)          /*Loop for the stars*/
11         printf("*");              /*Prints a star*/
12      printf("\n");
13     }
14    return 0;
15  }
```

Output:

```
Enter the number of lines to print:  5
    *
   **
  ***
 ****
*****
```

Each line in the above pattern consists of one or more blank spaces followed by one or more stars. For a given line the total number of blank (b) and star (s) is equal to the number of lines to print (n). i.e. **b+s=n**.

So the number of blank spaces is equal to b=n-s. Here the number of stars in a given line is equal to the line number `i` for that line. Hence we replace s by `i` to get the number of blanks in a given line as b = n-i. The nested `for` loop in line-8 is used to print the blanks. The variable `j` is used to count the blanks and hence goes from `1` to `n-i` for the line number `i` as derived from the outer for loop.

The next example prints the pyramid pattern shown on the right for number of lines=4:


Printing Pyramid

```
*
***
*****
*******
```

```
1  /*Program-64: Using nested for loops to print pattern-2*/
2  #include<stdio.h>
3  int main()
4  { int n, i, j, k;
5    printf("\nEnter the number of lines to print: ");
6    scanf("%d", &n);
7    for( i=1; i<=n; i++ )          /*Outer loop counts number of lines*/
8     { for(j=1; j<=(n-i); j++)     /*Loop for blanks in front of the stars*/
9         printf(" ");              /*Prints a blank space*/
10     for(k=1; k<=2*i-1; k++)      /*Loop for the stars*/
11        printf("*");              /*Prints a star*/
12     printf("\n");
13    }
14   return 0;
15 }
```

Output:

```
Enter the number of lines to print:  5
    *
   ***
  *****
 *******
*********
```

The code for the above pattern is similar to the last one with the only difference in line-10. If you observe you will find that the number of stars in a given line forms an A.P. series 1, 3, 5, 7, 9 etc. The ith term of this series is 2*i-1. Hence the `for` loop of line-10 repeats from `1` to `2*i-1`.

The next example prints the inverted pyramid pattern shown on the right for number of lines=4:


Printing Inverted Pyramid

```
*******
 *****
  ***
   *
```

```
1  /*Program-65: Using nested for loops to print pattern-2*/
2  #include<stdio.h>
3  int main()
4  { int n, i, j, k;
5    printf("\nEnter the number of lines to print: ");
6    scanf("%d", &n);
7    for( i=n; i>=1; i-- )          /*Outer loop counts number of lines*/
8     { for(j=1; j<=(n-i); j++)     /*Loop for blanks in front of the stars*/
9         printf(" ");              /*Prints a blank space*/
10     for(k=1; k<=2*i-1; k++)      /*Loop for the stars*/
11        printf("*");              /*Prints a star*/
12     printf("\n");
13    }
14   return 0;
15 }
```

Output:

```
Enter the number of lines to print:  4
*******
 *****
  ***
   *
```

The only change in the above code from program-64 is in the `for` loop of line-7. Instead of starting from `i=1` we have started from `i=n` as we have to print the base of the pyramid first. The loop counter is thereafter

decreased from `n` to `1` by decrementing the loop counter by `i--` in the `for` loop header. As the loop counter decreases from `n` to `1`, the loop condition checked is `i>=1`.

The next program calculates the total number of tea, coffee or cold drink items sold from a coffee counter by prompting the user for each sell.

```
1   /*Program-66: Beverage vending software using a do-while loop and switch-case*/
2   #include<stdio.h>
3   int main()
4   { char prompt;
5     int tea=0, coffee=0, drink=0;
6     printf("******** Beverage Vending Shop ********\n");
7     do{ printf("\nEnter T (Tea), C (Coffee), D (Cold Drink) and E (Exit): ");
8         fflush(stdin);
9         prompt = getchar();
10        switch(prompt)
11            { case 't' :
12              case 'T' :   tea++;
13                           break;
14              case 'c' :
15              case 'C' :   coffee++;
16                           break;
17              case 'd' :
18              case 'D' :   drink++;
19                           break;
20              case '\n':
21              case 'e' :
22              case 'E' :   break;
23              default  :   printf("\nEnter a valid choice!");
24            }
25      } while( ! ((prompt=='e')||(prompt=='E')) );
26    printf("\nTotal Sell: Tea=%d, Coffee=%d, Cold Drink=%d", tea, coffee, drink);
27    return 0;
28  }
```

Inner switch block

Outer do-while block

Output:

```
******** Beverage Vending Shop ********
Enter T (Tea), C (Coffee), D (Cold Drink) and E (Exit): t
Enter T (Tea), C (Coffee), D (Cold Drink) and E (Exit): c
Enter T (Tea), C (Coffee), D (Cold Drink) and E (Exit): c
Enter T (Tea), C (Coffee), D (Cold Drink) and E (Exit): d
Enter T (Tea), C (Coffee), D (Cold Drink) and E (Exit): T
Enter T (Tea), C (Coffee), D (Cold Drink) and E (Exit): D
Enter T (Tea), C (Coffee), D (Cold Drink) and E (Exit): c
Enter T (Tea), C (Coffee), D (Cold Drink) and E (Exit): C
Enter T (Tea), C (Coffee), D (Cold Drink) and E (Exit): d
Enter T (Tea), C (Coffee), D (Cold Drink) and E (Exit): e
Total Sell: Tea=2, Coffee=4, Cold Drink=3
```

In the above program, the user inputs the type of drink, i.e. tea, coffee or cold drink, into the variable `prompt` in line-9. The variable `prompt` is then used for checking the option entered by the user, by placing it as the condition variable for the `switch` statement in line-10.

If the user inputs say '**C**', the value in `prompt` is '**C**'. This value is then checked with each of the `case` constant values i.e. '**t**', '**T**', '**c**', and finally when the `case` constant '**C**' is encountered in line-15, the condition gets satisfied and the statements following '**C**' get executed i.e. the value of the variable `coffee` is incremented by one and the control `breaks` out of the `switch` block only.

After this the **while** condition is again checked in line-7 and the process repeated till the user enters 'e' or 'E', when the **while** condition gets false and the program prints the total sell and quits.

Note how the character **case** constants have been used. For example for a **coffee** item, as both 'c' and 'C' can be a valid input, the **case** condition in line-14 gets true for 'c'. Once a match has been found, the **control simply falls to the next statement** in the absence of any statements to execute following the case 'c', irrespective of whether the following **case**s have matched or not. This is **similar to a logical OR operation** in an **if** condition. The **break** statement in line-16 finally takes the program control out of the **switch** block.

```
10      switch(prompt)                    prompt='c'
11      { case 't' :
12        case 'T' :    tea++;
13                      break;
14        case 'c' :
15        case 'C' :    coffee++;
16                      break;
```

Control will go directly from the case 'c' to the coffee++ line, bypassing the case 'C' statement

Also note how the **do-while** condition in line-25 works. If the user enters either 'e' or 'E' in the prompt, the condition part **((prompt=='e')||(prompt=='E'))** gets True. But the **NOT** operator (**!**) placed before the condition makes the overall condition False, thereby stopping the loop.

However, if the user enters any other character in the prompt, **((prompt=='e')||(prompt=='E'))** becomes False. Now the **NOT** operator (**!**) placed before the condition makes the overall condition True, thereby continuing with the loop.

One can do away with the **fflush(stdin)** statement in line-8 as the same has been taken care of in line-20 by the **case** '\n' statement.

C to it that.....

The following examples show the areas where mistakes are common and some better ways to deal with a problem. The whole program is not written always and only the relevant parts are shown.

The following program section uses the logical and increment/decrement operators to evaluate an expression as true or false. Note the way in which the expression is evaluated:

```
1   int main()
2   { int a=0, b=0, c=2, d;
3     c = ++a && ++c || ++b;
4     d = a && b && c;
5     printf("\nValue of a=%d", a);
6     printf("\nValue of b=%d", b);
7     printf("\nValue of c=%d", c);
8     printf("\nValue of d=%d", d);
9     return 0;
10  }
```

Output:

```
Value of a=1
Value of b=0
Value of c=1
Value of d=0
```

If the output of the above program segment surprises you, here is the reason why it is such. As mentioned earlier, the value of the variable **c** is **evaluated from left to right**. As an AND operation has a higher priority than an OR operation, the AND operation will be carried out first. Hence first **a** is incremented to 1 by **++a**. The next operator is AND. Hence to test the logic, **c** is next incremented to 3 by **++c**. The result of **logical ANDing** of **(++a && ++c)** thus gives **(1 && 3) = True = 1**.

The next operator is OR. But the **first parameter of the OR operator has already been calculated to be** `1` and hence is sufficient to declare the final output `c` as logically true [since, (1) OR (any-value) = True = 1]. Thus further calculation of the remaining portion of the expression is not carried out and hence `b` **is not incremented to** `1` **by** `++b` and remains as `0`. The final value of `c` thus becomes `True` or `1`. Hence finally `a = c = 1`.

Next when `d` is calculated, `a && b` gives a logical `0` as `b = 0` and hence the expression is not tested any further as the logical output of ANDing will always give a `0` in this case as the first expression is a `0`. The final value of the variable `d` is thus equal to `0`. This explains the above outputs.

The increment operator can give rise to some tricky problems as shown below:

```
1   int main()
2   { int a = 1, b, c;
3     b = a++ + ++a;
4     c = a+1;
5     printf("\nValue of a=%d", a);
6     printf("\nValue of b=%d", b);
7     printf("\nValue of c=%d", c);
8     return 0;
9   }
```

Output:

```
Value of a=3
Value of b=4
Value of c=4
```

The expected output for `b` is `b=1+2=3`. But the actual output will `be b=2+2=4`. This is because `++a` **will be calculated first** and will make `a=2`, then this will be added to `a` of `a++`. **After** the addition, `a++` will make `a=3`. Then **this** `a` will be added to `1` and make `c=3+1=4`.

The following programs show how the increment operators are evaluated when placed within a function:

```
1   int main()
2   {int x=2, y=3;
3    printf("\nx=%d, y=%d, x=%d", ++x,  y++,  x++);
4    printf("\nx=%d, y=%d",  x,  y);
5    return 0;
6   }
```

Output:

```
x=4, y=3, x=2
x=4, y=4
```

The output of line-3 is due to the reason that `printf()` **reads the variables starting from the right to left**. Hence the `x++` expression is evaluated first. Since this is a post increment operation, the `printf()` function first prints the current value of `x` i.e. `2` and then increments it to `3` by the `x++` operation. Next it prints the current value of `y` i.e. `3` due to the same reason and increments it to `4` by the `y++` operation. Finally it encounters the `++x` expression. This being a pre-increment operation, `x` is first incremented from 3 to `4` and then the value is printed. Line-4 simply prints the current values of `x` and `y` i.e. `4` and `4`.

```
1   int main()
2   {int x=2, y=3;
3    printf("\nx=%d, y=%d, x=%d", x++, y++, x++);
4    printf("\nx=%d, y=%d", x, y);
5    return 0;
6   }
```

Output:

```
x=3, y=3, x=2
x=4, y=4
```

The functioning of the above program piece is similar to the previous program.

Now consider the following program section.

```
1   int main()
2   { int i=1;
3     while(i<=100);
4        { printf("%d\n", i);
5          ++i;
6        }
7     return 0;
8   }
```

The above loop will **run infinitely** without showing any output, in other words it will simply go on running when executed. The reason lies in the placing of a semicolon (`;`) at the end of the `while` condition.

This is a **common programming mistake**. This implies there are no segments to execute in the body of the loop and if the condition is true, the control will come back to test the condition again. But since the variable `i` has not been incremented, the condition will remain true always, `i` being equal to the initial value of `1`. Another reason why a loop can run infinitely is when the condition is **true always** like `while(1)`. The rectification is shown below:

```
3     while(i<=100)
```

Some other expressions for infinite loops are given below:

```
for (;1;);
for (;;);
while (1);
do{ } while(1);
```

The **program-54** of page P1-11-20 is repeated below with a better method that is **more efficient in performance** than program-54. Calculate the value of the exponential series e^x given by the relation: $e^x = 1 + x/(1!) + x^2/(2!) + x^3/(3!) + x^4/(4!) + \ldots$ to n terms.



More efficient method to find e^x using last term method

```
1   int main()
2   { int n, i;
3     float e = 1.0, x, power = 1.0, factorial = 1.0;
4     printf("\nInput the value of x: ");
5     scanf("%f", &x);
6     printf("\nInput number of terms to include: ");
7     scanf("%d", &n);
8     for(i=1; i<n; i++)
9        { power = power * x;
10         factorial = factorial * i;
11         e = e + power/factorial;
12       }
13    printf("\nThe value of e^x up to %d terms is %f", n, e);
14    return 0;
15  }
```

Output:

```
Input the value of x: 1
Input number of terms to include: 10
The value of e^x up to 10 terms is 2.718282
```

In the above method, we have not used a nested `for` loop as in program-54. In program-54, each time a new term is calculated, the `power` i.e. x^n and the `factorial` value i.e. n! are calculated anew. In the

above example however, **we are using the previous term already calculated, to calculate the current term**, with a slight modification. This reduces the total calculation time and hence **increases the efficiency** of the program.

For example to calculate the value of e^x up to 11 terms, using the method of program-54 we would have to do a total of (1+2+3+4+... +10) = 55 multiplications for numerator and 55 multiplications for denominator i.e. a total of 110 multiplications in all (for the first term i.e. 1, we would not have to do any multiplication).

However using the above method we have to do 2 multiplications for each term (one for the `power` and one for the `factorial`) i.e. a total of 2*10 = 20 multiplications for the 11 terms (for the first term i.e. 1, we would not have to do any multiplication). Therefore **the total number of multiplications is much less**. This will be more prominent when the number of terms increases.

In line-3 the variables `e`, `power` and `factorial` are initialised to `1`. The `for` loop starts from line-8.

Initial starting value of **power=1.0, factorial=1.0, e=1.0**

Run1: i=1, power = power*x = 1*x = x, factorial = factorial*i = 1*1 = 1 = 1!
e = e + power/factorial = 1 + x/1!

Run2: i=2, power = power*x = x*x = x^2, factorial = factorial*i = 1*2 = 2 = 2!
e = e + power/factorial = 1 + x/1! + x^2/2!

Run3: i=3, power = power*x = x^2*x = x^3, factorial = factorial*i = 2*3 = 6 = 3!
e = e + power/factorial = 1 + x/1! + x^2/2! + x^3/3!

Run4: i=4, power = power*x = x^3*x = x^4, factorial = factorial*i = 6*4 = 24 = 4!
e = e + power/factorial = **1 + x/1! + x^2/2! + x^3/3! + x^4/4! etc.**

Thus we find that **with each run of the loop we get a new term** of the series by the process shown above without doing the calculation from scratch to calculate the `power` and `factorial`.

The program to calculate the **value of sin(x)** given by the series sin(x) = $x - x^3/3! + x^5/5! - x^7/7! + \ldots$ up to n terms is repeated with a more efficient method than program-55.



Calculating sin(x) using last term method

```
1   int main()
2   { int n, i, sign = -1;
3     float sin, x, y, power, factorial;
4     printf("\nInput the value of the angle in degrees: ");
5     scanf("%f", &y);
6     printf("\nInput number of terms to include: ");
7     scanf("%d", &n);
8     x = 3.14159*y/180;
9     power = x;
10    factorial = 1;
11    sin = power/factorial;
12    for(i=2; i<n; i++, sign *= -1)
13       { power = power*x*x;
14         factorial = factorial*(2*i-1)*(2*i-2);
15         sin += sign*power/factorial;
16       }
17    printf("\nThe value of sin(%f) up to %d terms is %0.3f", y, n, sin);
18    return(0);
19  }
```

Output:

```
Input the value of the angle in degrees: 30
Input number of terms to include: 10
The value of sin(30.000000) up to 10 terms is 0.500
```

In this method, the calculations are done similar to the previous problem, i.e. **using the previous term to calculate the current term**. However here the `power` and `factorial` part of the successive terms form an A.P. series 1, 3, 5, 7, ... whose i^{th} term is given by **2*i-1**.

Note that except for the first term, all the other terms are calculated by taking help of the previous term. Therefore the **first term is calculated separately in line-11** and the loop starts from line-12 with `i=2` and goes up to `n-1`.

Each subsequent `power` term is calculated by multiplying the previous power by x^2 in line-13. Similarly each new `factorial` is calculated by multiplying the previous `factorial` by the terms `(2*i-1)*(2*i-2)`.

To see how it works, consider `i=4` for which `(2*i-1)*(2*i-2) = (2*4-1)*(2*4-2) = (8-1)*(8-2) = 7*6`. Now from the previous term i.e. term number-3 we have the previous value of `factorial = 5!` Therefore for `i=4`, the `factorial = factorial*(2*i-1)*(2*i-2) = 5!*7*6 = 7!` as required.

Line-8 of the program is used to **convert the angle entered** by the user from degree to radians.

Similar to infinite `while` and `do-while` loops we can also have an infinite `for` loop as stated earlier. The following example shows the use of an infinite `for` loop.



Infinite *for* loop

```
1   int main()
2   { int count=1;
3     printf("\nProgram to print all even numbers from 1 to 100:");
4     for(;;)
5       { if(count>100)
6             break;
7       if(count%2==0)
8           printf(("\n%d", count);
9       count++;
10    }
11    return 0;
12  }
```

The **condition part** of the `for` loop **does not contain any parameter**, however the **semicolons ";" must be given**. The `if` statement in line-5 is used to check the condition and the `break` statement in line-6 is used to break out of the loop once the condition in line-5 gets satisfied. Therefore when the value of the variable `count` becomes more than 100 by the `count++` operation in line-9, the condition in line-5 becomes True and the loop terminates.

One can have **all types of possibilities** like `for(i=1;;)` or `for(;n<100;)` or `for(;;i++)` or `for(i=0;;i++)` etc. while dealing with a `for` loop, but the **semicolons are a must**. To exit from such a loop, the condition is checked using an `if` statement along with a `break` statement.



The Fact File

- **Repetition of a code in C is done using the while, do-while and for statements**
- a++ is same as a=a+1, the ++ operator is called the increment operator
- **a-- is same as a=a-1, the -- operator is called the decrement operator**
- The meaning of the statement x+=y is the same as x=x+y. += is called a compound operator. Other compound operators include -=, *=, /=, %=
- **The effect of these statements on the variable y is the same, though the final count value is different**
  y = count+1;
  y = ++count;
- The increment operator ++ when placed in front of a variable is known as pre-increment operation. When pre-increment takes place, the variable is first incremented and then it is used. Therefore y=++a; is same as a=a+1; y=a;
- **The increment operator ++ when placed after a variable is known as post-increment operation. When post-increment takes place then the variable is first used and then it is incremented. Therefore y=a++; is same as y=a; a=a+1;**

- **If an expression contains both a++ and ++a, then the ++a will be evaluated first followed by a++. Example if a=4, then y=a++ + ++a; will calculate the value of y as 10**
- +++x, x+++, or y---, ---y does not have any meaning by itself, though, z = x+++y does carry a meaning and is the same as z = (x++) + y
- **In a while loop, each time the loop begins to execute, the condition is tested first and if found true, the body of the loop is executed. Else if the condition becomes false, the control goes out of the loop and executes the statement next to the loop.**
- The while loop is called an entry controlled loop as the condition is checked before entering the loop body
- **The while loop is more suited for execution of a loop unknown number of times, depending on certain calculations or user input.**
- Example of while loop:
```
while( x++<100 )
    printf("\n %d", x);
```
- **If continuous sum operations are used in a loop (like sum = sum+term), the sum variable needs to be initialised to 0 (zero) before the loop starts. If continuous product operations are used in a loop (like product = product*term) then the product variable needs to initialised to 1 before the loop starts**
- The do-while loop is similar to the while loop in that it continues as long as the condition remains true. But the main difference is that, while the while loop may or may not execute at all depending upon the condition, the do-while loop will execute at least once before testing the condition as the condition in a do-while loop is tested at the end of the do-while block
- **The do-while loop is called an exit controlled loop as the condition is checked at the end of the loop**
- To test for a valid input using a do-while loop, the condition of the do-while loop should be just the reverse of the required input.
- **The do-while loop is used for situations where a code needs to be executed at least once, like finding the hcf or gcd**
- Example of do-while loop:
```
do { puts("Enter +ve no.");
     scanf("%d", &num);
   } while (num < 0);
```
- **The most important and widely used of the loop structures is the *for* loop. It can be used in place of the *while* or *do-while* loops with proper modification.**
- The *for* statement executes in the following stepwise manner. The loop counter is first set to initial value. Then the condition is tested. If condition is found true then body of loop is executed. Next the counter is incremented/decremented. The condition is again tested and if found true then body of loop is executed. The process is repeated till condition becomes false.
- **The for loop is usually used in situations where the number of repetitions is pre stated or defined**
- E.g. of for loop:
```
for( x=0; x<100; x++ )
    printf("\n%d", x);
```
- **By the term nesting of loops we mean placing one loop within another. Any type of loop can be placed within any other type of loop.**
- The comma operator in general allows two expressions to appear at a place, where otherwise only one expression would have been used. Example for (i=1, j=9; i<20; ++i, --j)
- **If the comma operator is used to separate two or more conditions in the loop header, then only the last condition will be evaluated**
- The for loop can be used with no argument in the loop header to all combinations of arguments as shown below:

```
for ( ; ; )          /*The three parts may be defined elsewhere, but semicolons are must*/
for ( i=1; ; )       /*Only the initialisation of loop counter done in loop header*/
for ( ; i<10 ; )     /*Only condition checking done in loop header*/
for ( ; ; ++i )      /*Only re-initialisation or increment of loop counter done in loop header*/
for ( i=1; i<10; )   /*Initialisation of loop counter and condition checking done in loop header*/
for ( ; i<10; ++i )  /*Condition checking and increment of loop counter done in loop header*/
for ( i=1; ; ++i )   /*Initialisation of loop counter and increment of loop counter done in loop header*/
```

- **The loop statement for (; ;); forms an infinite loop that will run for ever**
- The break statement is used to forcibly come out of a program loop bypassing the normal loop condition test. The program control comes out of the loop and the statement immediately next to the loop gets executed.
- **The continue statement is used to take the control to the beginning of the loop, bypassing certain statements within the body of the loop.**

Review Questions

Q1. **Multiple Choice Questions. Select from any one of the four options.** **1 each**

i) Which of the following is not a proper operation in C?
a. y=x++; b. y=+x+; c. y=++x; d. y=+x++;

ii) What is the final value of y in the C code: x=2; y=++x; y=x++;
a. 2 b. 3 c. 4 d. 5

iii) What is the final value of y in the C code: x=2; y=x++; y=++x;
a. 2 b. 3 c. 4 d. 5

iv) Which of the following is not a compound operator in C?
a. -= b. %= c. ?= d. *=

v) Which of the following is not a loop control structure in C?
a. do-while b. for c. while-do d. while

vi) Which one of the following is an exit controlled loop control in C?
a. for b. while c. do while d. none of these

vii) How many arguments are there in the loop header of the while loop?
a. 1 b. 2 c. 3 d. 4

viii) How many arguments are there in the loop header of the for loop?
a. 1 b. 2 c. 3 d. 4

ix) Which of the following codes will not form an infinite loop?
a. while(2>1); b. do{ }while(-2); c. for (; 1;); d. while(2-3 > 3-2);

x) Which command is used to forcibly come out of a loop in C?
a. continue b. case c. default d. break

xi) Which loop structure is used if you want to repeat a code at least once?
a. for b. while c. do while d. none of these

xii) How many times will the following loop run? *int x=4; while(--x){ puts("Good Luck"); }*
a. 2 b. 3 c. 4 d. 5

xiii) How many times will the following loop run? *int x=4; while(x--){ puts("Good Morning"); }*
a. 2 b. 3 c. 4 d. 5

xiv) How many times will the following loop run? *int x=4; while(x-1){ --x; }*
a. 2 b. 3 c. 4 d. 5

xv) The following loop will run how many times?

```
int x=5; y=2;
do { if(y=2)
        printf("\n%d", x); } while(y- -);
```

a. 1 time b. 2 times c. Infinite times d. None of these

xvi) What will be the output of the following C code?

```
int z, y=3;
z = y + 5 + ++y;
printf("%d, %d", y, z);
```

a. 11, 4 b. 4, 12 c. 4, 13 d. 3, 12

xvii) What will be the output of the following C code?

```
int x=2;
printf("%d, %d, %d", x, x++, ++x);
```

a. 4, 4, 3 b. 2, 2, 2 c. 2, 2, 4 d. 4, 3, 3

xviii) What will be the output of the following C code?

```
int x=5; printf("%d, %d, %d",++x, x, x++);
```

a. 7, 6, 5 b. 5, 5, 5 c. 6, 6, 6 d. 5, 6, 7

xix) What will be the output of the following piece of code in C?

```
int x=6;
while(- -x) printf("%d," , x);
```

a. 4, 3, 2, 1, b. 5, 4, 3, 2, c. 5, 4, 3, 2, 1, d. 6, 5, 4, 3, 2, 1,

xx) What will be the output of the following piece of code in C?
int y, x=5; y = 2 + x++ + x; printf("%d, %d", x, y);
a. 5, 14 b. 5, 12 c. 6, 12 d. 6, 14

xxi) The following C code will display the output:
int x=3;
while(- -x) printf("%d,", x);
a. 2, 1, b. 3, 2, c. 3, 2, 1, d. 2, 1, 0

xxii) How many times will the following C code run?
int x=-5;
while(2+x) { x++; }
a. 1 b. 2 c. 3 d. 4

xxiii) The following loop in C will run how many times?
int x=5; y=5;
while(x = y) { printf("%d,", x); y- -; }
a. 1 time b. 3 times c. 5 times d. none of these

xxiv) What will be the output of the following code in C?
int x=3;
while(- -x) printf("%d,",--x);
a. 3, 2, 1, b. 2, 1 c. 1 d. none of these

xxv) How many times will the following C code run?
char x='Z';
do { printf("\n%c", x);
x--; } while (!x>='z');
a. 26 b. 0 c. 1 d. 32

xxvi) How many times will the following C code run?
int p=3, q=5;
while ((p>q)?(p=p-q):(q=q-p))
printf("%d", q);
a. 1 b. 2 c. 4 d. 3

xxvii) How many times will the following C code run?
int count=1;
while (count>=-10 && count<=10)
{ printf("Count=%d", count);
count++; }
a. 40 b. 30 c. 10 d. 20

xxviii) How many times will the following C code run?
int x=3, y=5;
while(x-y) printf("%d", x++);
a. 1 b. 2 c. 3 d. 4

xxix) What will be the output of the following code in C?
int p=4;
do { p = p/++p; p--; } while(p);
printf("%d", p);
a. 3 b. 1 c. 0 d. 2

xxx) How many times will the following C code run?
int p=4, q=3;
do { if(q=1) printf("\n%d",p);} while(--q);
a. 1 b. 2 c. 3 d. 4

Q2. Short Answer type questions: **1 each**

i) Name two entry controlled loop structures in C
ii) State one difference between a while loop and a do-while loop.
iii) Write an expression for an infinite loop using the *for* loop structure in C.
iv) Write an expression for an infinite loop using the *while* loop structure in C.
v) Write an expression for an infinite loop using the *do-while* loop structure in C.
vi) What is the use of the ++ operator in C?
vii) What is the meaning of the compound operator *= in C?
viii) What is the equivalent statement using compound operators for the statement *y = y % z;*
ix) What is the utility of the *continue* statement in C?
x) What is the function of the break statement in C?
xi) How many times will the following loop run? *int x=5, y=8; while(x-y) printf("%d", x++);*
xii) How many times will the following loop run? *int x=5, y=8; for(; x>0, y>0; --x, --y) puts("Bye");*
xiii) What is the meaning of the statement *x=++y;* in C?
xiv) Is there any difference between x++ and ++x in C?
xv) What is the use of the comma operator in C?

Q3. Long Answer type questions: **7 each**

i) Write a program to find the factorial of a number using a do-while loop. Explain the meaning of post-increment operation in C. Write any two compound operators in C. 4+2+1

ii) Write a program to check if an input number is prime or not. Explain the meaning of pre-increment operation in C. What is the use of comma operator in C? 4+2+1

iii) Write a program to find the hcf or gcd of two numbers. Explain the difference between pre-increment and post-increment operation in C with the help of an example. 4+3

iv) Show the use of the continue statement in C with the help of a proper code example. State two differences between the continue and break statements in C. How many different compound operators are available in C? 4+2+1

v) Write a program to print only the nth term of a Fibonacci series, where n is a user input. Compare the three loop structures while, do-while, and for. 4+3

vi) What do you mean by a nested loop? State two differences between a while and a do-while loop. State the difference between the statements y=a+1; and y=++a; Name one situation where the do-while loop is more suitable to use. 2+2+2+1

Q4. Assignment Programs: **4 each**

i) Write a program to print the terms and find the sum of n terms of a G.P. series where the first term, common difference and the number of terms to print are entered by the user.

ii) Write a program to print the sum of the first n terms of the following series: 10, 8, 6, 4, 2, ...

iii) Write a program to print the from term number a to term number b of the following series, where a and b are two integer values entered by the user (a<b): Series = 3, -9, 27, -81, ...

iv) Write a program to check if a number entered is such that all the digits in the number are odd in nature [example, 153179, here all the digits are odd]

v) Write a program to check if a number entered is such that all the digits in the number are even in nature [example, 24628, here all the digits are even]

vi) Write a program to check if a number entered is such that all the digits in the number are alternately odd and even in nature starting from the right [example, 234181, 1=odd, 8=even etc.]

vii) Write a program to check if a number entered is such that it contains no zeroes in it. If zeroes are present then count the number of zeroes present [example, 234181 has no zeroes but 201406 has two zeroes.]

viii) Write a program to check if a number entered is such that consecutive digits in the number are in ascending order starting from the right [example, 86531, here 1<3<5<6<8]

ix) Write a program to check if a number entered is such that the consecutive digits in the number are in descending order starting from the right [example, 24589, here 9>8>5>4>2]

x) Write a program to check if a number entered is such that the consecutive digits in the number are also consecutive in nature [example, 65432, here 2, 3, 4, 5, 6 are consecutive digits]

xi) Write a C program to find the sum of the digits at the even position and the sum of the digits at the odd position (starting from right) of an integer number separately and print the sums. [Example: for the number 162457, sum_odd=7+4+6=17, and sum_even=5+2+1=8]

xii) Write a C program to find the sum of the even digits and the sum of the odd digits of an integer number separately and print the sums. [Example: for the number 162457, odd=7+5+1=13, and even=4+2+6=12]

xiii) Write a C program using a while loop to check if a number entered is an Armstrong number.

[If the sum of cubes of each digit of a 3 digit number is equal to the number itself, it is called an Armstrong number. E.g. $153 = 1^3 + 5^3 + 3^3$]

xiv) Write a C program to check if a number entered is a perfect number i.e. if the sum of its factors is equal to the number itself. [For example the factors of 6 are 1, 2, 3 and 6 = 1+2+3, the factors of 28 are 1, 2, 4, 7, 14 and 28 = 1+2+4+7+14]

xv) Write a C program to find the sum of the factorial of the digits at the even position and the sum of the factorial of the digits at the odd position (starting from right) of an integer number separately. Compare the sums and print the larger of the two sums. [Example: for the number 1624, sum_odd=4!+6!, and sum_even=2!+1!]

xvi) Write a C program to print all the Armstrong numbers between 100 and 999. Note: Armstrong numbers between 100 and 999 include 153, 370, 371, and 407.

xvii) Write a program in C which is used to repetitively add the digits of a number till the resultant sum is a single digit number. Then check if the final sum is equal to 1. [For example: $9253 \Rightarrow 9+2+5+3 = 19 \Rightarrow 1+9 = 10 \Rightarrow 1+0 = 1$]

xviii) Write a program to find and print all prime numbers between A and B, where A and B are two integer values entered by the user, such that A<B.

xix) Write a C program to find whether or not a number n input through the keyboard is a factorial number or not using a for loop. For example the number 120 is a factorial number and is equal to 5! But the number 100 is not a factorial number.

xx) Write a C program using a do-while loop to calculate the simple interest for a given principal. The program should have the following features:

- The program should input from the user, the Principal, Rate of interest and Time of investment.
- The program should check for valid inputs and give error messages in case of invalid (i.e. negative) values.
- The program should ask for multiple runs and will terminate if '0' is entered as the Principal.

xxi) The coefficient of the rth power of x for the binomial expansion of $(1+x)^n$ is given by:

$^nC_r = n!/[r! * (n-r)!]$

Write a C program to calculate the coefficient of x^r. The user should input n and r.

xxii) Print the terms and find the sum of the following series up to n terms:

a. S1 = 2, 5, 8, 11, 14, ...
b. S2 = 4, 8, 16, 32, 64, ...
c. S3 = 2/3, 4/5, 8/7, 16/9, ...
d. S4 = 1, 1, 1, 3, 5, 9, 17, 31, ...
e. S5 = 1, 2/2!, 3/3!, 4/4!, 5/5!, ...
f. S6 = 1, 1/(3!), 1/(5!), 1/(7!), 1/(9!), ...

xxiii) Print the terms and find the sum of the following series up to n terms (here 0<x<=1):

a. $x - x + x - x + x - x + \ldots$
b. $x - x^2 + x^3 - x^4 + x^5 - \ldots$
c. $x^2 - x^5 + x^8 - x^{11} + x^{14} - \ldots$

xxiv) Find the sum of the following series up to 'n' terms for a value of x as input by the user.

$$S = 1 - \frac{3x}{9*1!} + \frac{6x^4}{10*3!} - \frac{9x^7}{11*5!} + \frac{12x^{10}}{12*7!} - \ldots$$

xxv) Write a C program to evaluate: $\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \ldots$ Where the angle x is in radians and is input by the user. The series is calculated up to n number of terms as input by the user.

xxvi) Find the sum of the following log series up to 'n' terms where (0<x<=1)

$$\log(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \ldots$$

xxvii) Print the following patterns using loops (the number of lines 'n' to print following the pattern is input by the user):

a.
```
*
**
***
****
***** ... ...
```

b.
```
***** ... ...
****
***
**
*
```

c.
```
    *
   ***
  *****
 *******
********* ... ...
```

d.
```
    *
   ***
  *****
   ***
    *
```

xxviii) Print the following patterns using loops (the number of lines 'n' to print following the pattern is input by the user):

a.
```
    *
   **
  ***
 ****
***** ... ...
```

b.
```
***** ... ...
 ****
  ***
   **
    *
```

c.
```
*********
 *******
  *****
   ***
    * ... ...
```

d.
```
*****
** **
*   *
** **
*****
```

xxix) Print the following patterns using loops (the number of lines 'n' to print following the pattern is input by the user):

a.
```
1
22
333
4444
55555 ... ...
```

b.
```
55555 ... ...
 4444
  333
   22
    1
```

c.
```
    1
   123
  12345
 1234567
123456789 ... ...
```

d.
```
      1
     121
    12321
   1234321
  123454321
```

xxx) Print the following patterns using loops (the number of lines 'n' to print following the pattern is input by the user):

a.
```
A
BB
CCC
DDDD
EEEEE ... ...
```

b.
```
9
87
654
3210
```

c.
```
    1
   232
  34543
 4567654
567898765
```

d.
```
       1
      101
     10001
    1000001
  100000001 ... ...
```

xxxi) Write a program to generate a pattern as shown below, where the variable n is entered by the user. The value of n should be greater than or equal to 3. The patterns for n=3, n=4, and n=5 are shown below.

n=3

```
  *
 * *
* * *
 * *
  *
* * *
```

n=4

```
   *
  * *
 * * *
* * * *
 * * *
 * *
 * *
* * * * *
* * * * *
```

n=5

```
    *
   * *
  * * *
 * * * *
  * * * *
* * * * * *
  * * * * *
  * * * * *
  * * * * *
  * * *   * *
  * * *   * *
  * * *   * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
```

xxxii) Write a program to generate a pattern as shown below, where the variable n is entered by the user. The value of n should be greater than or equal to 2 (ensure that the user enters n>=2). The patterns for n=2, n=3, and n=4 are shown below.

n=2

```
  1   1
1 2 1 2 1
  1
```

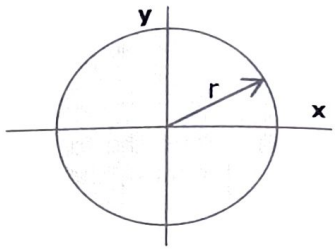
n=3

```
    1
  1 2 1     1 2 1
1 2 3 1   1 2 3 2 1
    2 1
    3 2 1
```

n=4

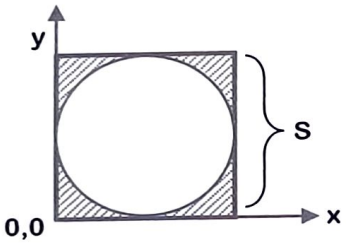
```
      1
    1 2 1
  1 2 3 2 1     1 2 3 2 1
1 2 3 4 3 2 1   1 2 3 4 1
    2 1
    3 2 1
```

xxxiii) From the figure shown on the right find all the points that are present inside and on the circle in the first quadrant. The radius of the circle is a user input. Print the points as x, y pairs like (1,3), (2,3) etc.

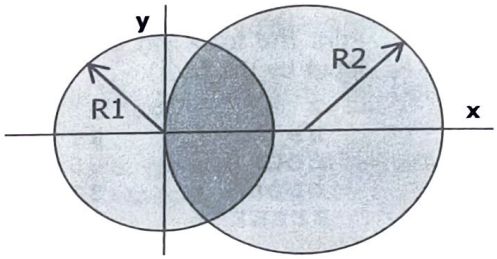


xxxiv) From the figure of the last problem find all the points that are present inside and on the circle in all the quadrants. The radius of the circle is a user input.

xxxv) Write a program to find all points that are present inside the shaded area of the diagram shown below. The bottom left corner of the square is aligned with the origin. The square has a side equal to S that is a user input.



xxxvi) In the diagram shown on the right, the circle with radius R1 is centred at the origin and the circle with radius R2 touches the y axis. It has its centre on the x axis. Write a program to find all points that are present at the common shaded region of the overlapping circles. R1 and R2 are user inputs. Take R1<R2.



CHAPTER 12
Functions in C

- Introduction 12-1
- The Working of a Function 12-2
- Defining a Function 12-3
- Function Recursion 12-10
- Storage Classes 12-12
- Some worked out examples 12-22

12.1 Introduction

When a program becomes large, it may be difficult to organise everything within a single main function. Suppose the main( ) function consists of several thousand lines of code. Within such a program it will be very difficult to locate any portion of the code that may need debugging or change and will be similar to finding a needle in a haystack! Moreover some portion of the code may have to be repeated at different points of the same program leading to rewriting of the same code at the different points. For example if the factorials of several numbers need to be calculated at several places, we have to write the same code to do the calculation at all those places. This would unnecessarily increase the size of the program.

A **Function** is a block of instructions that together can perform a certain specific task.

Under such circumstances, if the program could have been **broken down into several separate logical compartments based on the type of job carried out** by each section of the code, then it would have been easier to handle the program. The feature that C provides to do this type of coding is called a function.

A function is a block of instructions that together can perform a certain specific task. As had been mentioned at the beginning of the course, a C program in general consists of a number of such functions or modules of code; `main()` being the most important one. You had been already using several library functions to carry out such jobs as taking inputs from the keyboard [`getchar()`, `scanf()` etc.], printing output onto the screen [`printf()`, `puts()`], finding the square-root of a number [`sqrt()`] etc. These functions were **available in the standard library provided by C** and one had to just **call these functions to execute them**. Each of these functions had a unique identifiable purpose and executed a specific type of job as had been defined by the scope of that particular function. Thus to print something onto the screen at several points in a program, instead of writing the whole code at all those places, one just calls the `printf()` function to get the desired output.

What is Function

Depending upon the way a function is available, there are **two different categories of functions**:

- **Library functions**: These are **pre-written and pre-compiled functions** and stored in the **standard library of C**. The functions are written for doing specific jobs like printing some text on the screen, finding a square root, allocating memory at run time etc. To use these functions, the programmer has to include specific header files and call the functions from the program. Examples of such functions are `printf()`, `scanf()`, `puts()`, `sqrt()`, `malloc()`, `strlen()`, `strcpy()`, etc.
- **User defined functions**: It is highly unlikely that functions to do all types of jobs are available in the standard libraries. Every program has its own set of special or unique necessities leading to the use of some special functions, which may not be available in the standard libraries. These functions are written by users for doing specific tailor made jobs as per the user requirement. To use these functions, the **user has to first define the function** i.e. type the code for the function and then call the function to carry out its specific tasks. We will now discuss the methods to define and use a user-defined function. Before doing that, let us write down the usefulness of using functions in a program.

Library and User-Defined functions

Library functions are pre-written and pre-compiled functions stored in the standard library of C.

The utilities of using functions:

a. **Repetitive jobs can be assigned to a function**: Any repetitive job that needs to be executed at several places of the program can be assigned to a particular function. For example consider the sine series given by the relation $\sin(x) = x/(1!) - x^3/(3!) + x^5/(5!) - \ldots$ Here the factorial of a number is to be calculated at several points of the program and necessitates the use of a function called factorial which can be called each time a factorial needs to be calculated.

b. **Increases readability of a program**: A function consists of self-contained components i.e. it is self sufficient in nature. Thus if a program is compartmentalised using functions to carry out different jobs, it becomes easier to follow the program logic as one has to concentrate on a certain portion of the code only. This increases readability of a program.

Utility of functions

c. **Reduces chances of error and helps in debugging**: Increase in readability also helps in pin pointing possible errors within a section of the code and accordingly helps in debugging.

d. **Helps in shared coding**: A large program consisting of several functions can be developed by several persons. Each person can code one or more functions independently and then all the modules can be put together to build the whole program later on. This makes writing of large programs faster as the whole job can be broken down into several logical parts, which can be simultaneously coded.

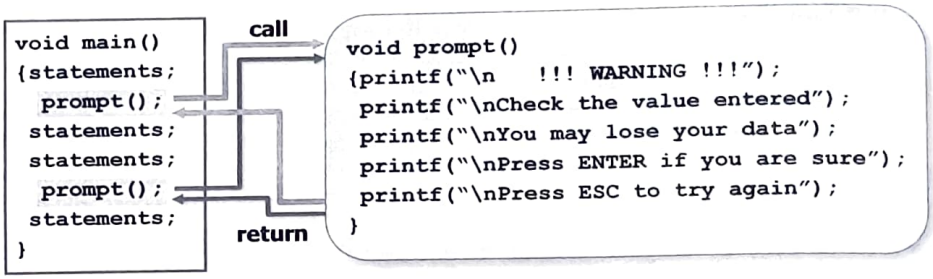
e. **Reuse of code**: A function carrying out a certain particular work can be reused in more than one program by making libraries of such user-defined functions and calling them in the programs as and when required. This saves a lot of time unnecessarily rewriting the same code in separate programs.

12.2 The Working of a Function



Working of a function

Let us now see how a function works. The diagram on the right shows how a user defined function called `prompt()` is called from the `main()` function twice and gets executed each time it is called.



When the function `prompt()` is called for the first time, the program control temporarily goes from `main()` function to the called function i.e. `prompt()`. After execution of the statements under `prompt()`, the control again comes back to the statement next to the function call (follow the grey arrows). The same process is repeated again, the next time the function `prompt()` is been called (follow the black arrows).

Though the above example is of no important use but it illustrates the way a function is used in a program. Instead of writing the code for the **common task** of printing on the screen the same set of lines at several points in the `main()` function, we have used a function called `prompt()` to do the same job for us. Whenever the message needs to be printed we just call the function `prompt()` and it executes the job for us. Later on if you want to do some extra thing using `prompt()`, you just add the extra lines inside the body of the function `prompt()` once, and it will work as per your requirement.

```
int main()
{statements;
   printf("\n   !!! WARNING !!!");
   printf("\nCheck the value entered");
   printf("\nYou may lose your data");
   printf("\nPress ENTER if you are sure");
   printf("\nPress ESC to try again");
 statements;
 statements;
   printf("\n   !!! WARNING !!!");
   printf("\nCheck the value entered");
   printf("\nYou may lose your data");
   printf("\nPress ENTER if you are sure");
   printf("\nPress ESC to try again");
 statements;
}
```

On the other hand if you had not used a function to do the job then **you would have to write the same code at all the places** where you needed those lines of text, as shown in the figure on the right. Moreover in case you needed a **change at a later stage, you would have to rewrite the changes at all those points** where you had used that piece of code. Surely using a function is a much better option!

A function is similar in structure to the `main()` function. Just like the `main()` function it processes some information when requested for. **The request to a function is referred to as calling the function**. But before being called, the **function needs to be defined first**.

A function can be defined at two different places as described below:

- In the first method a **function prototype** (discussed later) is declared before writing the `main()` function and the function defined at a later stage. Declaring a prototype results in informing the compiler in advance during compilation process the function name and other function parameters so that it can cross check with the correctness of the function calls made at a later stage in the `main()` and other

functions. Moreover, when the compiler encounters the function calls later on during the compilation, it knows in advance that these refer to functions that are defined later on. **If the compiler encounters a function call before it has been declared or defined, then it causes a program error** and displays the message that "function prototype not found".

Figure 1 shows the general structure of a program using functions whose prototypes are declared before the `main()` function, and the functions defined after `main()`.

The **alternative method** is to **define the functions first** and then **use them later** in `main()` or other functions. Figure 2 shows the use of this method. In this case the compiler already has the definitions of the functions before they are being called by `main()` (or other functions) and hence does not cause any error during compilation.

```
include header files

prototype for function_1() declared;
prototype for function_2() declared;

main()
 { statements;
     function_1() called;
   statements;
     function_2() called;
     function_1() called;
   statements;
 }

function_1() defined
 { statements for function_1; }

function_2() defined
 { statements for function_2; }
```

Fig. 1

```
include header files

function_1() defined
 { statements for function_1; }

function_2() defined
 { statements for function_2; }

main()
 { statements;
     function_1() called;
   statements;
     function_2() called;
     function_1() called;
   statements;
 }
```

Fig. 2

Of these two methods, the **first method is the preferred one** because in that case a function once declared as a prototype can be defined anywhere i.e. either before or after the `main()` function (but never inside the `main()` function, as a **function definition within another function is forbidden**).

Whereas in the second method, the functions need to be defined before `main()`, keeping in mind the fact that the functions should be so arranged that a function is not been called before it has been defined. This is to be particularly noted for the case when a user-defined function calls another user-defined function. Thus in the above example if `function_1()` calls `function_2()`, then instead of defining `function_1()` first and then `function_2()`, we have to define `function_2()` first and then `function_1()`.

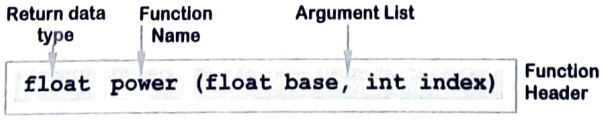
But this situation can be completely avoided if function prototypes are used. The functions being declared first, it does not matter the order in which they are defined later.

(Note: A **function can also be declared within the `main()` function** along with other variable declaration. However, remember that **a function can never be defined within another function**)

12.3 Defining a Function

Irrespective of whether a function is declared using a prototype or not, let us discuss how a general function is to be defined. **A function definition consists of**:

The function Header: This consists of the function return data type, the name of the function, and the function argument list (or parameter list).



The return data type: A function may do some calculations (for example calculate the power of a number) and then supply this calculated value to the point from where the function was called. The 'return data type' of a function indicates the data type (i.e. `int`, `float`, `char` etc.) of this calculated value that the function sends back to the point from where it was called.

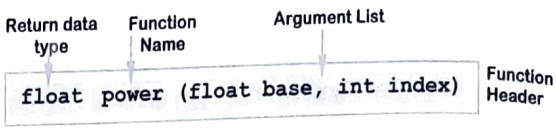
A Function should be defined outside any function definition. A function cannot be defined within another function.

A Function definition consists of a head and a body. The head has a return data type, name, and parameter list.

Function header

The data passed to a function for its use form the Function Argument List.

- **The function name**: Each function is identified by a name (like `printf`, `scanf` etc.). The function name follows the same rules applicable for identifiers or variable names. Thus valid function names include `power`, `average`, `root1`, `root2`, `_1stPos`, `SimpleInterest`, `VectorProduct` etc.



- **The argument list**: When a function is called to do some calculations, it may be supplied with some values with which it is supposed to do the calculations. This list of values forms the argument list or **parameter list** of a function. For example when the `power()` function shown above is called, one has to supply both the base and the index to the function to calculate the power i.e. $base^{index}$. Thus the variables `base` and `index` form the argument list of this function.

Function body

- **The function Body**: It consists of the code that the function executes to carry out its specific job. It starts after the function header and is enclosed within a pair of curly braces.

The following example will clarify the meaning and the working of a function. In the following program an user-defined function called `power()` is used to find the value of x^y where `x` is a `float`, and `y` is an integer and both are input by the user. The values of `x` and `y` are entered by the user in the `main()` function and then they are utilised by the user-defined function to calculate the desired output and send the outcome of the calculation back to the main function.

Example of an user defined function

```
1  /*Program-67: Example of a User Defined Function and Function Call*/
2  #include<stdio.h>
3  float power(float base, int index)
4    { float result=1;
5      while(index--)
6          result = result*base;
7      return result;
8    }
9  int main()
10   { float x, output; int y;
11     printf("\nEnter base x (decimal value): ");
12     scanf("%f", &x);
13     printf("\nEnter index y (integer value): ");
14     scanf("%d", &y);
15     output = power(x,y);
16     printf("\nValue of x^y is %f", output);
17     return 0;
18   }
```

The function Header

The Body of the function

Function Definition

Function call

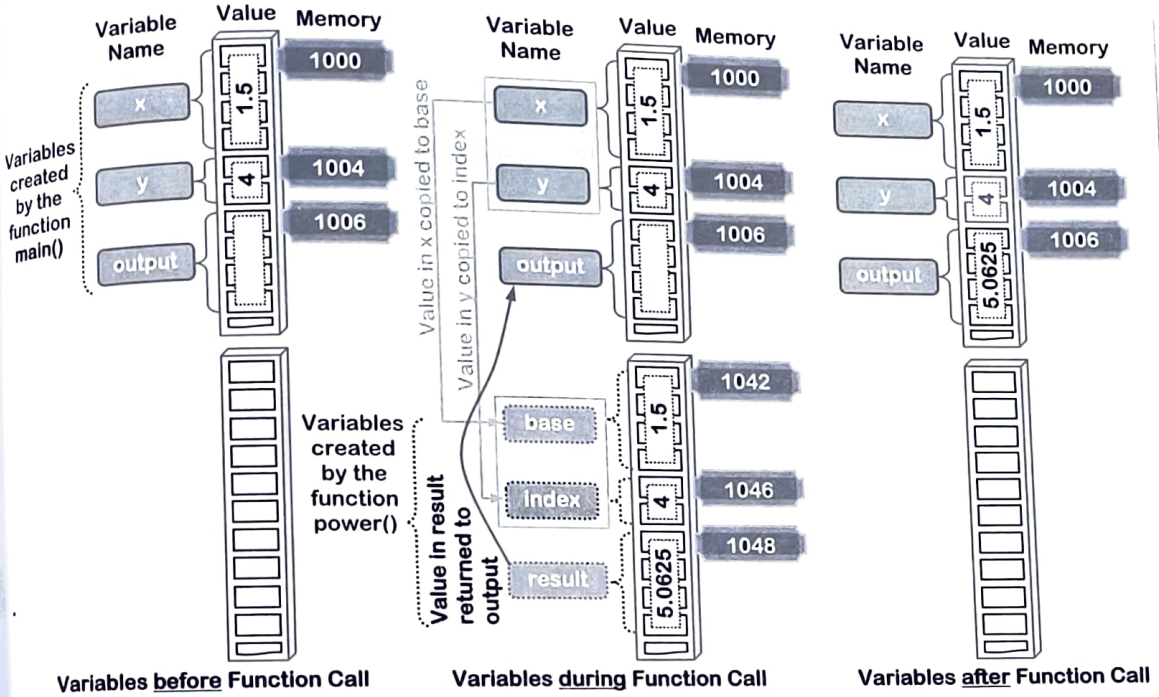
Let us now analyse the above example. First, let us consider the `main()` function from where the program starts. Line-10 declares two `float` type variables, `x` and `output`, and one `int` type variable `y`. The **first memory diagram** in the next page shows how **memory space** can be allocated for storing these variables declared under the `main()` function.

The user enters the two values in `x` and `y` in lines-12 and 14 respectively. Let the user input the value `1.5` for `x` and `4` for `y`. Therefore the memory location `1000` will now contain the value `1.5` and the memory location `1004` will contain the value `4`.

These variables that are passed from the calling function to the called function are called the actual arguments.

Next, in line-15 the user-defined function `power()` is called. For the function `power()` to work, it should know what is to be raised to what power. This information is supplied by placing the variables `x` and `y` inside the braces after the function name `power()`, as shown in line-15. Each such variable or parameter is to be written separated by a comma.

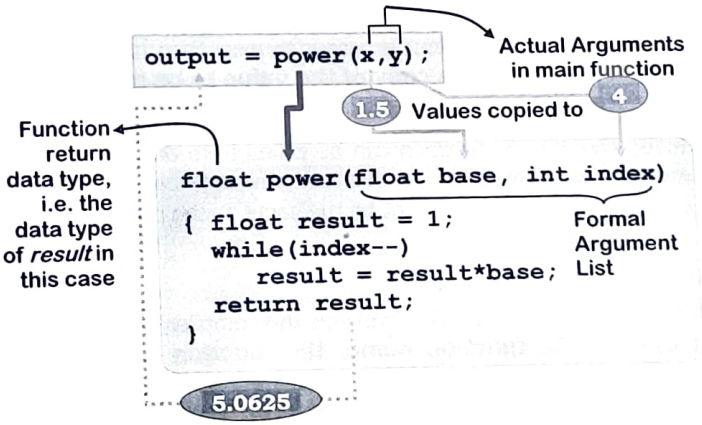
These **variables that are passed from the calling function to the called function are called the actual arguments**. Thus in this example the variables `x` and `y` are the actual arguments.



Memory diagram during function call

Let us now understand step by step how the user defined function will respond to this.

- When the function is called, the values stored in the actual arguments are **copied and transferred** to the argument list variables in the function header. The figure on the right shows how these values are transferred to the function `power()` from `main()`.



Working of a function call

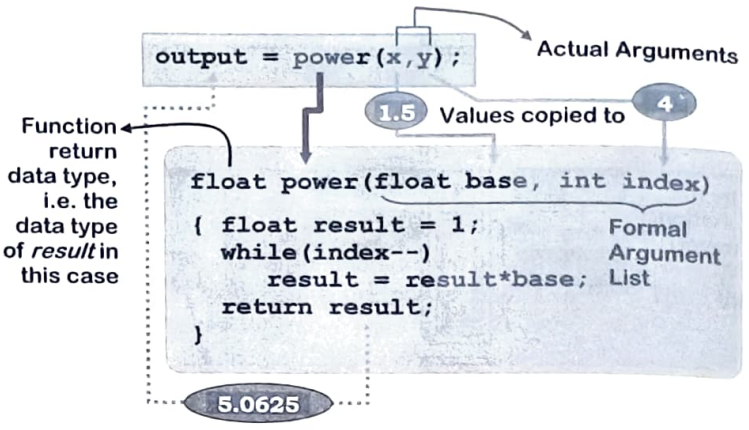
- First memory space is allocated for the variables `base` and `index`. Next the value stored in the variable `x` is **copied** to the variable `base` and the value stored in the variable `y` is copied to the variable `index`. **These variables that receive the values in the function header are called formal arguments**. Thus `base` and `index` form the formal arguments in this example. The middle memory diagram shows the memory allocation for the various variables during the function call. Note that **memory space is allocated for storing the formal arguments, only after the function is called**.

Formal Arguments

Variables in the function header that receive the values passed to the function are called formal arguments.

- The formal arguments should be **placed in the same order** inside the function parameter list as the actual arguments. That is, the first actual argument `x` is a `float` and so is the first formal argument `base`. Similarly the second actual argument `y` is an `int` and so is the second formal argument `index`. **The number, type and order of formal arguments must match the number, type and order of the actual arguments**, for the function to work properly.
- In this example the `float` value `1.5` is passed to the `float` variable `base` and the `int` value `4` is passed to the `int` variable `index` (refer to the **middle memory diagram**).
- The code, that the function is going to execute, is placed within a pair of curly braces `{}` just as in the `main()` function. It forms the body of the function.
- For doing its internal calculation, the function declares the variable `result` as `float` and assigns value '1' to it in line-4 (similar to variable declaration within `main()` function). Note that memory space is allocated for storing the variable `result`, **after** the function is called (refer to middle memory diagram).

- The `while` loop starting in line-5, then calculates the required `power`. With each run of the loop, the variable `result` gets multiplied by the `base` value. The variable `index` acts as the loop counter. With each `index--` the value in `index` gets decremented by 1 and in this way when `index` becomes 0, the loop condition becomes false and the loop terminates. For example in this case for `index=4`, the loop will run 4 times before `index` becomes 0 and will produce the `result` as $(1.5)^4 = 5.0625$.



- After the calculations the keyword `return` is used to send the value stored in the variable `result` to the calling function `main()`. The statement `return result` sends the value of the calculated `float` variable `result` i.e. `5.0625` to the `float` variable `output` in `main()`. The data type `float` written before the name of the function indicates that the data type of the `return` value is a `float`. The return value is then assigned to the variable `output` in `main()`. One thing should be ensured that **the data type of `output` should be the same as the data type of the return value**.
- The third memory diagram in the previous page indicates the memory allocation for different variable **after the function call**. Note that the variable `result` is local to the function `power()` i.e. the variable is created each time the function is called and **ceases to exist after the function** has completed its execution. Thus the value stored in `result` is also lost after the function is exit. The question that may rise is that if the variable ceases to exist then how the calling function can know the value. The answer lies in the fact that a **copy of the value to be returned is made**, and this copy is made available to the return point in the program.

Note: Though any function can be called from any other function, **a function cannot be defined inside another function**. Thus, in this case `power()` cannot be defined inside `main()`. But the order in which the functions are defined may not be the same as the order in which they are called.

prototype

Declaring a Function prototype:

A prototype for the function provides the compiler with the basic information about how the function is used. It indicates the **function name, the function argument data types and names**, and the **function return data type**. The prototype declaration has the same syntax as the function definition header with the exception that a **semicolon** is added at the end of the header to distinguish it from a function definition. We write below the previous program by using a function prototype. Note that since a prototype has been declared, it becomes unnecessary to define the function before it is been called by `main()` and hence is generally defined after the closing braces of the `main()` function.

A prototype for the function provides the compiler with the basic information about how the function is used and its nature.

As is evident from the piece, the function prototype for the user-defined function `power()` i.e. `float power(float base, int index);` indicates the following:

- The **return data type** of the function `power()` is a `float` i.e. the function returns a `float` value
- The **name** of the function is `power`
- The **argument list** consists of a `float` type data variable with name `base` and an `int` type data variable with name `index`

After the declaration of the function prototype in line-3, when the function is being called in the `main()` function in line-10, the two values `x` and `y` are passed to `base` and `index` and the calculations are done as before and finally the result returned to the `main()` function.

When using several functions, generally the function prototypes of all the functions are stated before `main()` and then defined one by one after the `main()` function.

```
/*Program-68: Example of a Function Call with Function Prototype*/
#include<stdio.h>
float power(float base, int index);
void main()
  { float x, output; int y;
    printf("\nEnter base x (decimal value): ");
    scanf("%f", &x);
    printf("\nEnter index y (integer value): ");
    scanf("%d", &y);
    output = power(x,y);
    printf("\nValue of x^y is %f", output);
  }
float power(float base, int index)
  { float result=1;
    while(index--)
        result = result*base;
    return result;
  }
```

The function Prototype declared. Note the semicolon after the declaration.

Function called at this point

NO semicolon after the function header for definition

The function Header

The Body of the function

Function Definition

The return statement:

As was mentioned earlier, the **`return`** statement is **used to send back control to the calling function**. It also returns any value (can be result of any calculation) that is placed after the keyword **`return`**. The value returned may be any **literal or can be any variable name or an expression** that evaluates to a certain value. However remember that **a function in C can return a single value only**. More than one value cannot be returned by a function (however you can return multiple values using a structure type data).

The following examples show different uses of the **`return`** statement:

```
/*Program-69: Example of a function returning a constant value*/
int bonus(int x)
  { printf("\nYour bonus score is %d", x/10);
    return 100;
  }
```

The above function called **`bonus()`**, receives an **`int`** type data (received in the variable **`x`**). It calculates the bonus score and displays the bonus in line-3. Finally the function returns a fixed value **`100`** in line-4.

```
/*Program-70: Example of a function returning a calculation*/
double molecules(float moles)
  { printf("\nThe total number of moles of the gas = %f", moles);
    return (6.022e23 * moles);
  }
```

In the above program, the function called **`molecules()`** receives a **`float`** type data (received in the variable **`moles`**) in line-2. It prints the number of moles in line-3 and finally **calculates the number of molecules present** by multiplying the number of moles with the Avogadro Constant 6.022×10^{23}. Since the result of the calculation is a **`double`** type value, the return data type has also been declared as a **`double`**. The calculated result is then directly returned by the function to the calling function.

```
/*Program-71: Example of a function returning a calculation*/
float TotalPressure(float p1, float p2)
  { printf("\nThe Partial Pressures are %f, and %f", p1, p2);
    return (p1+p2);
  }
```

In the above program, the function called **`TotalPressure()`** receives two **`float`** type data (received in the variables **`p1`** and **`p2`**) in line-2, that represent two different partial pressure values. It prints these partial pressure values in line-3 and finally calculates the total pressure as the sum of the two partial pressures.



The 'return' statement

The return statement sends control back to the calling function and may return a value also.

A function can return a single value only at any given time.

A function can return any type of data.

Since the result of the calculation is a `float` type value, the return data type has also been declared as a `float`. The calculated result `(p1+p2)` is then directly returned by the function to the calling function.

Function returning logical value

```
1  /*Program-72: Example of a function returning a logical value*/
2  int Check(int x, int y)
3    { printf("\nMistakes can fetch you only lesser points!!");
4      return ((x==1)&&(y==1));
5    }
```

In the above program, the function called `Check()` receives two `int` type data (received in the variables `x` and `y`) in line-2. It prints a comment in line-3 and finally returns the logical output of the calculation `(x==1)&&(y==1)`. For example, if the value of `x` is `1` and that of `y` is `0`, then the logical output of the above expression becomes false, and the `return` statement returns the value `0` for false. On the other hand, if `x` is `1` and `y` is also `1`, then the expression evaluates to true and `return` statement returns the value `1` for true.

Function returning a calculated value

```
1  /*Program-73: Example of a function returning a calculated value*/
2  float volume(float p, float t)
3    { float vol;
4      vol = 8.1*t/p;
5      return vol;
6    }
```

In the above code, the function called `volume()` receives two `float` type data (received in the variables `p` and `t`) in line-2. It declares a variable called `vol` in line-3 that it uses to calculate the volume of the gas with pressure `p` and temperature `t`. In line-4, the required volume is calculated and assigned to the variable `vol`. In line-5, the value stored in the variable `vol` is returned to the calling function.

A function may use local variables for its own use. These need not be passed to the function as argument.

Note that in line-3, the function declares the `float` type variable `vol` for its **own use**. This variable is **NOT received** by the function in the argument list. For any internal calculation, **a function may declare and use local variables**. The return data type of the function is `float`, as the data type of the variable returned i.e. `vol` is `float`.

Function returning a constant character

```
1  /*Program-74: Example of a function returning a constant character*/
2  char yes()
3    { printf("\nCORRECT Answer!!");
4      return 'Y';
5    }
```

The function called `yes()` receives nothing in its argument list. It prints a comment in line-3. In line-4, the function returns a constant character `'Y'`. Accordingly the function return data type is declared as a `char`.

Function returning a character

```
1  /*Program-75: Example of a function returning an input character*/
2  char Input()
3    { char x;
4      printf("\nEnter the Section in which you study: ");
5      x = getchar();
6      return x;
7    }
```

In the above function, the function receives nothing in its argument list. In line-3 a `char` type variable `x` is declared for use by the function. In line-4 the user is asked to enter a 'section' value. In line-5, the `getchar()` **function is used to input the section** value from the user into the variable `x`. This value entered by the user is then returned by the `return` statement in line-6.

The function `maximum()` shown next, receives two `int` type values in the variables `x` and `y` in line-2. In line-4, using the conditional operator, the maximum between the two values is calculated and returned by the `return` statement.

```
/*Program-76: Example of a function returning maximum amongst 2 values*/
int maximum(int x, int y)
   { printf("\nThe maximum of the two values returned");
     return (x>=y)?(x):(y);

   }
```

In all the above cases the function returned certain values to the calling function, be it an `int`, a `float`, a `char` or a `double`. As mentioned earlier, the type of data returned by the function is mentioned at the beginning of the function definition or declaration.

In case no such thing is mentioned, then **the default data type returned by a function is always an** `int`. Thus in case a function returns a `float` value and the return data type is not mentioned before the function name, then correct data will not be transferred to the calling function as shown below:

The default return data type of a function is an int type data.

```
/*Program-77a: Example of a function without an explicitly declared return data type*/
cube(float x)
   { return (x*x*x);
   }
```

The above function is used to find the cube of a `float` variable. The variable to cube is passed to the function called `cube()` which does the cubing and sends back the result via the return statement. If someone inputs the value of say `x=5` then the return result will be `125` as expected. But if someone enters `5.5` then instead of getting `166.375` one will get the result as `166`. The reason is that, return type of the function being not declared explicitly, it is taken by default as an `int`. Hence the **function returns only the integer portion of the result**. To rectify the problem, the function `cube()` is to be declared as a `float`:

```
/*Program-77b: Example of a function with a declared return data type*/
float cube(float x)
   { return (x*x*x);
   }
```

If no return data type is mentioned, a function returns the data as an integer, even if the original data is of another type

Now the return data type being declared as a `float`, the correct result i.e. `166.375` will be returned by the function, when the value of `x` is `5.5`.

In all the above examples, each function had only one return statement. Though a function can return a single value, but in general **a function can contain as many return statements as required. However in that case only one return statement will get executed finally**. Moreover it is not necessary that the `return` statement should be placed at the end of the function; it can be placed anywhere in the function.

The following program demonstrates the above point. It inputs a character and uses a function to check if it is a lower case character. If so, it converts it to upper-case, else it prints out the original character.



Changing from lower to upper case

```
/*Program-78: Function Call to change lower to upper character*/
#include<stdio.h>
char toupper(char letter);
void main()
   { char lower, upper;
     printf("\nEnter any character:");
     scanf("%c", &lower);
     upper = toupper(lower);
     printf("\nThe upper case letter is %c", upper);
   }
char toupper(char letter)
   { if(letter >='a' && letter <= 'z')
         return (letter-32);
     else
         return (letter);
   }
```

A character is input in line-7. In line-8 the function `toupper()` is called with the argument `lower`. The function `toupper()` receives the character within the variable `letter` in line-11. The function `toupper()` then uses the **ASCII value of the received** character `letter` to check if it is in lower or in upper case. If

found to be in lower case, i.e. if `(letter>='a' && letter<='z')` is true in line-12, it converts it to uppercase by the expression `(letter-32)` in line-13 and returns the value. In case the character is already in uppercase, it returns the unchanged character via the second `return` statement in line-15.

Output1

```
Enter any character:a
The upper case letter is A
```

Output2

```
Enter any character:B
The upper case letter is B
```

In general, whenever branching is there and the output depends on which program segment the program has branched to, the **different branches can each contain a `return` statement** to return a particular value.

It is **not necessary that a function should always return a value**. A function may print some messages or carry out a calculation or print a result without passing any value to the calling function. Under such cases the keyword **void** is placed before the function name in place of the return data type. **No return statement is used in such a function**; otherwise it will give error message during compilation. A modified form of the maximum value finding function is given below to show the use of the keyword **void**.

Function to find maximum value

```
1  /*Program-79: Function to find and display the maximum value*/
2  #include<stdio.h>
3  void MaxValue(int x, int y);
4  void main(void)
5    { int a, b;
6      printf("\nEnter number1: "); scanf("%d", &a);
7      printf("\nEnter number2: "); scanf("%d", &b);
8      MaxValue(a,b);
9    }
10 void MaxValue(int x, int y)
11   { int z;
12     z = (x>y)?(x):(y);
13     printf("\nThe maximum of the two numbers = %d", z);
14   }
```

It is not necessary that a function should always return a value

In the above example, the prototype for the function `MaxValue()` is declared in line-3 and defined in line-10. It has been defined to **return nothing i.e.** `void`, as the maximum value calculated by the function is displayed in the `MaxValue()` function itself.

The return data type of a function not returning anything is void

Similarly the `main()` function has also been defined as `void` as it is not returning any value. Moreover as it is not receiving any value either i.e. any formal argument, the keyword `void` has been placed within the brackets after `main()` to indicate the **absence of any argument**. A function defined as `void` should not contain any `return` statement and hence the usual `return 0` statement is absent from `main()`.

Refer to the worked-out examples in page P1-12-16 to clearly understand how functions work.

When a function calls itself it is called a Recursive Function

12.4 Function Recursion

In C a function can be made to call itself. A **function that has been constructed in such a manner as to call itself is known as a recursive function**. This is sometimes called circular definition i.e. the process of defining something in terms of itself. For a function to be recursive, **it must satisfy two basic conditions**:

- **It must have an ending point or stopping condition**
- **It must make the problem simpler**

Requirements for function recursion

The simplest function to illustrate the principles of a recursive call is the function to calculate the **factorial of a number**. We had earlier calculated the factorial of a number by using the `for` loop. Now we will carry out the same function without using the `for` loop but using function recursion.

By definition a factorial is defined as:

- `factorial(1) = 1`
- `factorial(n) = n x factorial(n-1)`

The above two rules of a factorial satisfy the conditions for recursion i.e. the **first definition of the factorial serves as an ending point** while the second definition simplifies the problem as the calculation of factorial(n-1) is simpler than factorial(n). Armed with the above rules and definitions let us now write down the **recursive version of the function to calculate factorial of a number.**



Factorial of a number using recursion

```
1  /*Program-80: To calculate the factorial of a number using recursion*/
2  #include<stdio.h>
3  long int factorial(int num);
4  void main()
5  { int n; long int fact;
6    printf("\nEnter number to find factorial: "); scanf("%d", &n);
7    fact = factorial(n);
8    printf("\nThe factorial of %d is = %ld", n, fact);
9  }
10 long int factorial(int num)
11 { long int f;
12   if(num == 1)
13      return 1;
14   f = num * factorial(num-1);
15   return f;
16 }
```

Output

```
Enter number to find factorial: 4
The factorial of 4 is = 24
```

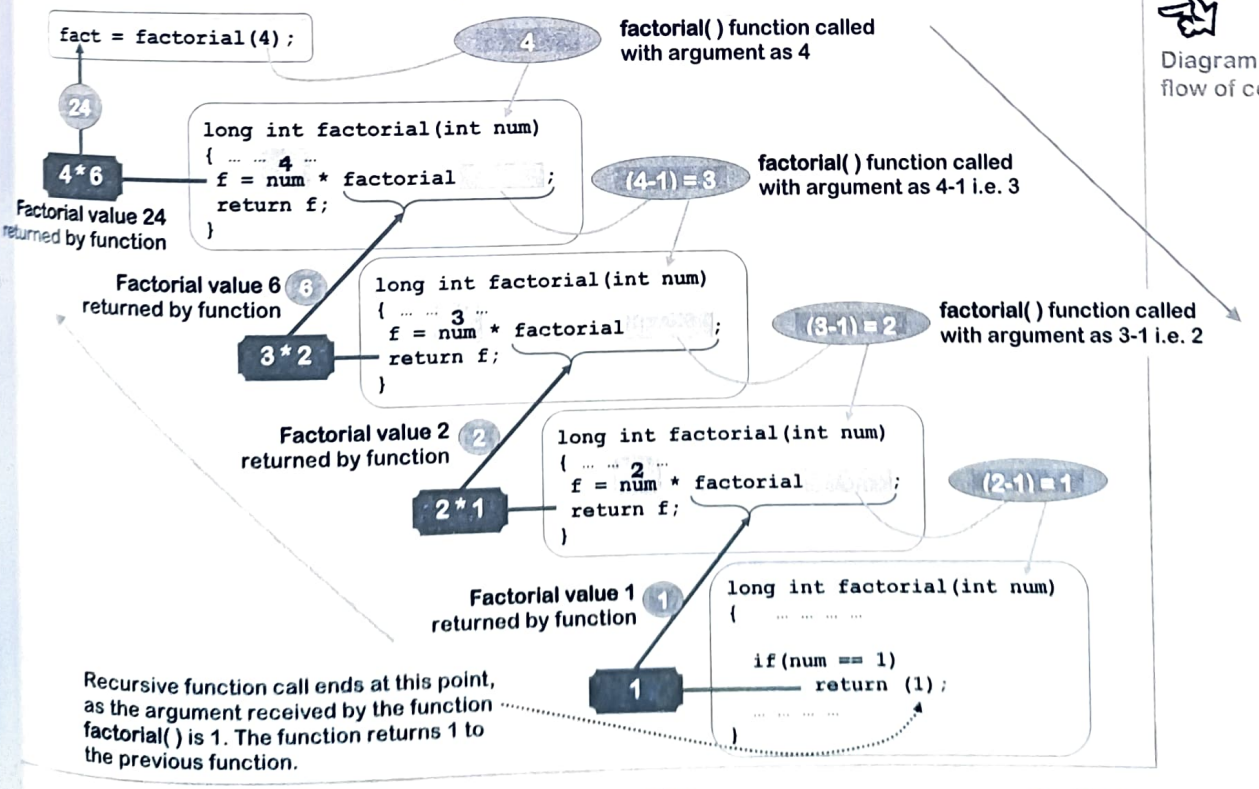
For recursion to take place the function must have a stopping condition.

The diagram below illustrates the flow of program control for calculating the factorial of 4 using the above recursive program. The `main()` function calls `factorial()` with `4` as its actual argument and the function in return returns back the calculated value for factorial 4. But before returning the computed value, the function again calls itself with the actual argument as (4-1) = 3 and expects a return value and waits. This second call to the function with 3 as the actual argument causes another call to the `factorial()` function with (3-1) = 2 as actual argument. Finally this leads to another self call with actual argument as (2-1) = 1.

At this point, the called function has an argument of '1' which satisfies the `if` condition in line-12 and hence returns '1' without executing any further self function calls. In this way the return chain continues until result of all the calls are returned back to the calling functions.



Diagram showing flow of control



The next program calculates the result of x^y using recursion. This can be done using recursion, as one can express x^y recursively as:

$$x^y = x * x^{(y-1)} = x * x * x^{(y-2)} = \ldots \ldots = x * x * x * \ldots x * x^{(y-y)}$$

Moreover $x^{(y-y)} = x^0 = 1$ can be taken as the stopping condition.

Calculating power using recursion

```
1   /*Program-81: To calculate the power of a number using recursion*/
2   #include<stdio.h>
3   float power(float x, int y);
4   void main()
5   { float base, output;
6     int index;
7     printf("\nEnter the base: ");
8     scanf("%f", &base);
9     printf("\nEnter the index: ");
10    scanf("%d", &index);
11    output = power(base, index);
12    printf("\nThe required value of %f to the power %d is = %f", base, index, output);
13  }
14  float power(float x, int y)
15  { if(y == 0)
16      return 1;
17    else
18      return x * power(x, y-1);
19  }
```

Output

```
Enter the base: 2
Enter the index: 3
The required value of 2.000000 to the power 3 is = 8.000000
```

The above function works similar to the last one. The terminating condition for the loop is when the **index** becomes equal to '0' i.e. for **power(x,0)**, as $x^0=1$. The **if** statement in line-15 checks this and stops the recursive call. For all other values of index **y**, the return value of the function is the product of the **base** times one less than the current **index**. The calculation of 2^3 shown below clarifies the working of the function.

```
power(2,3) = 2 * power(2,2)
           = 2 * 2 * power(2,1)
           = 2 * 2 * 2 * power(2,0)
           = 2 * 2 * 2 * 1 = 8          [as power(2,0)=1]
```

Though recursion is a useful tool, as it can sometimes make a program much more compact, but before going for recursion one should also check for other alternatives such as using a loop. A **program execution using loops can at times be efficient** in reducing overhead compared to recursive calls. This is because at each recursive call, the compiler may have to generate copies of the arguments to the function and keep track of the location to return to when each return statement is executed, adding to the memory overhead.

12.6 Storage Classes

Types of storage classes

In C **every variable has two attributes**. They are the **scope and the class** of the variable. By scope of a variable we mean the area of the program in which the variable is valid and by class of a variable we mean whether the variable is temporary or permanent.

The Scope of a variable means the area of the program in which the variable is valid.

Till the point we dealt with functions, we used variables only within the **main()** function. But the moment we defined other functions besides the **main()** function, we had variables defined both in the **main()** function and in the user-defined functions. Under such circumstances a variable can have several scopes or options i.e. it can be valid only within **main()** or only within the user-defined function or can be valid in both the functions. Moreover **if the storage class of the variable is not mentioned in the declaration, then the compiler assigns the default storage classes** based on the context in which the variable is used. Hence in all the previous programs, the storage classes were assigned the default values. Each variable occupies a portion of the memory to store its value. The variable's **storage class determines in which**

location of the computer, the values in the variables are stored i.e. whether in the Memory or in the CPU Registers (special memories within the CPU).

Based on the scope and class of the variables, they are divided into 4 different types. These are:

- **Automatic** or local storage class variables
- **Register** storage class variables
- **Static** storage class variables
- **External** or Global storage class variables

The **Class** of a variable means whether the variable is temporary or permanent

The table below gives a **comparative study** of the scope and class of the above four storage classes:

Comparison of variable scope and class

Storage Class	Storage	Default Initial Value	Scope	Life
Automatic (syntax: `auto`) (All data types)	Memory	Unpredictable Garbage value	**Local** to the block in which the variable is defined	Till the control remains within the block in which it is defined
Register (syntax: `register`) (Only integers)	CPU Registers	Unpredictable Garbage value	**Local** to the block in which the variable is defined	Till the control remains within the block in which it is defined
Static (syntax: `static`) (All data types)	Memory	Zero	**Local** to the block in which the variable is defined	Value of the variable persists between different function calls
External (syntax: `extern`) (All data types)	Memory	Zero	**Global**	As long as the program's execution does not come to an end

1. **Automatic Storage Class:** These variables can be defined at the beginning of any block of code immediately after the opening brace (i.e. after the **{**). They are said to have **local or block scopes**. This means that they are **defined only within the particular block** (defined by a pair of curly brackets **{ }**) in which they are declared. They are born at the time the program control enters that block and ceases to exist when the control comes out of that block. Accordingly the same variable name can be used in more than one block with no fear of the compiler mixing up which variable to use when. This is because the **current value of an automatic variable is the value of that particular variable in the current block**. The following function block demonstrates the use and scope of the automatic class. Note that when not initialised, the value of the variable displayed will be garbage and within each block the variable has its own value.

Automatic storage class

Working of Automatic storage class

```
1  /*Program-82: Working of automatic variables*/
2  void main()
3  { auto int count1, count2=20;
4    printf("\nOuter count1 (uninitialised)=%d", count1);
5    count1=10;
6    printf("\nOuter count1 (initialised)=%d", count1);
7    printf("\nOuter count2 (initialised)=%d", count2);
8      { auto int count1=30;
9        printf("\nInner count1=%d", count1);
10       count1 = count1 + 50;
11       count2 = count2 + 40;
12     }
13   printf("\nOuter count1=%d", count1);
14   printf("\nOuter count2=%d", count2);
15 }
```

Inner Block

Outer Block

Output

```
Outer count1 (uninitialised)=575
Outer count1 (initialised)=10
Outer count2 (initialised)=20
Inner count1=30
Outer count1=10
Outer count2=60
```

The variables `count1` and `count2` are declared in the outer block in line-3. The variable count1 is first un-initialised and count2 is initialised to 20 in line-3. When the local variable `count1` **is again declared in the inner block**, it **hides the original** `count1`. The inner `count1` ceases to exist the moment the control comes out of the inner block. Hence the next `printf()` displays the same initialised value of `count1` as 10, as was declared in the outer block unaffected by the `count1=count1+50` statement. However `count2`, which was declared in the outer block has changed in value by using a variable in the inner block. This was possible because, since the outer block encompasses the inner block, the variable declared at the beginning of the outer block is available in the inner block (unless a variable of the same name is declared in the inner block as with `count1`). Hence the value of `count2` is displayed as `60` i.e. `20+40`.

In case no class specifier is mentioned, the variable is assumed to be automatic, making the above declaration optional.

Register storage class

A static variable gets automatically initialized to 0.

2. **Register Storage Class:** These variables are **similar to auto variables** with the only exception that they generally **occupy the CPU register memory spaces** instead of the conventional memory. Operations using register variables are faster in comparison to conventional memory stored variables. But not all variables can be defined as register variables as the **allowed data types for these are only integers** (some compilers may also support character type variables).

   Program **loop counters** and similar variables which are **accessed frequently** are declared as register variables. But care should be taken as all registers are not available at all times for storing these variables. This however will not cause any problem in program running as, in case register memory is not available then the register variable is automatically changed to an automatic variable by C. The syntax is:

```
register int index;
```

Static storage class

3. **Static Storage Class:** These variables also have **scope within the block in which they are declared**, but they are **initialised only once** and their **default initial value is '0'**. Whereas for the previous types of variable classes the value stored in a variable is lost forever once the control comes out of the block in which it is defined, in case of static variables, once initialised, the **value persists even between different function calls.**

   This type of use can be particularly useful when the contents of a variable need to be accumulated between different function calls such as to count how many times a function has been called as shown below:

Working of static variables Declaring a function

```
1   /*Program-83: Working of static variables*/
2   void counting();
3   void main()
4    { int i;
5      for(i=1;i<=3;i++)
6         counting();
7    }
8   void counting()
9    { static int count;
10     printf("\nFunction counting() called %d time(s)", ++count);
11   }
```

A static variable retains its value between function calls

Output

```
Function counting() called 1 time(s)
Function counting() called 2 time(s)
Function counting() called 3 time(s)
```

In the above example, the function `counting()` is first called by `main()` through the **for** loop in line-6. With the first function call, the value of the variable `count` is initialised to `0` in line-9. For every subsequent function calls to `counting()`, the value stored in the variable `count` is incremented and printed. Since the function is called 3 times, the value of `count` is printed 3 times (`i=1 to 3`) resulting in the output shown above. The point to note is that the **value of the variable** `count` **is not lost between function calls** and is incremented and accumulated.

In place of static variables if the `count` in the function `counting()` was declared as an automatic variable and initialised to 0, then the value of count would not persist between function calls and each time the function is called it will get initialised to 0. Thus the resulting output would have been:

```
Function counting() called 1 times
Function counting() called 1 times
Function counting() called 1 times
```

As a general note of caution, **avoid using static variables unless one really needs them**. Since their values are kept in memory even if they are not used and they are alive throughout the life of the program, they may use up a lot of memory space unnecessarily.

4. **External Storage Class:** These variables differ from the variables we have discussed so far mainly with respect to their scope. They **have a global scope** i.e. they are not defined or available within a particular block of code but are **available throughout the program**. As such they are also defined outside all functions, including `main()`. External variables are usually defined before `main()`, just like normal variable declaration.


External storage class

As external variables are global in nature, they can be assigned a value in one function and the same value can be used in another function. We know that a function in general can return a single value. However using global variables, more than one value calculated in a function can be made available to the calling function. Moreover **arrays** declared as global variables need not be passed between functions. Thus when a variable is used by many functions in a program, it can be declared as an external variable, thus avoiding the need to pass the variable between functions. The following example shows one use of global variables:


Use of global variables

```
1  /*Program-84: Working of global variables*/
2  #include<stdio.h>
3  void RealRoots(float a, float b, float c);
4  float root1, root2;
5  void main()
6  { float a, b, c;
7    printf("\nEnter the coefficients of the quadratic equation ax^2+bx+c=0: ");
8    scanf("%f %f %f", &a, &b, &c);
9    RealRoots(a, b, c);
10   printf("\nRoots of the quadratic equation are: x1 = %f, x2 = %f", root1, root2);
11 }
12 void RealRoots(float a, float b, float c)
13 { float m, n;
14   if( (b*b-4*a*c)<0 )
15     { printf("\Equation has no real roots");
16       exit(0);
17     }
18   m = -b/(2*a);
19   n = sqrt(b*b-4*a*c)/(2*a);
20   root1 = m+n;
21   root2 = m-n;
22 }
```

Output

```
Enter the coefficients of the quadratic equation ax^2+bx+c=0: 1  1  -6
Roots of the quadratic equation are: x1 = 2.000000, x2 = 3.000000
```

The global variables root1 and root2 are declared in line-4 as global float type variables. The value of the roots are calculated within the function `RealRoots()` and assigned to the variables `root1` and `root2` in lines-20 and 21. Since the variables are declared as global variables, they need not be returned and the values assigned to them in `RealRoots()` are also available in the `main()` function, where they are printed.

A C program can be composed of **multiple files** which are saved under a common project head. The project is then build, whereby the individual files are compiled separately and the object files are linked together to form a single executable program. Each such file may contain one or more related function definitions. In case of such multi-file programs a function definition in a given file can be treated as external. Such external functions will be recognised throughout the program.

The following program explains the method.



Multi-file program and use of extern storage class

FILE1.C

```
1  /*Program-85: Using external functions in a program*/
2  #include<stdio.h>
3  extern void message(void);        /*External function declaration in FILE-1*/
4  void main()
5    { message();
6    }
```

FILE2.C

```
1  extern void message(void)         /*External function definition in FILE-2*/
2    { puts("Have a nice day ...");
3      return;
4    }
```

Output

```
Have a nice day ...
```

When the project containing the above two files is built and run, the output shown above gets displayed.

12.6 Some worked out examples:

In C, a function can be defined to perform a variety of jobs. It can be used to carry out some calculations, print out some results or simply print some heading. Dividing a program into a number of functions usually reduces the chances of an error and helps in program understanding and debugging. So use of as many functions as possible is a good programming practice. The following examples show the use of various types of user defined functions.



Function to convert from Fahrenheit to Celsius

```
1  /*Program-86: Function to convert from Fahrenheit to Celsius*/
2  #include<stdio.h>
3  float Convert(float Fah);
4  void main()
5    { float C, F;
6      printf("\nEnter Temperature in Fahrenheit Scale: ");
7      scanf("%f", &F);
8      C = Convert(F);
9      printf("\nRequired temperature in Celsius Scale is %f", C);
10   }
11 float Convert(float Fah)
12   { float Cel;
13     Cel=(5.0/9.0)*(Fah-32);
14     return Cel;
15   }
```

Output

```
Enter Temperature in Fahrenheit Scale: 212
Required temperature in Celsius Scale is 100
```

The `main()` function inputs the temperature in Fahrenheit scale in line-7. Then the function `Convert()` is called with the argument `F`. The function defined in line-11, accepts the value within the variable `Fah`. It then calculates the required temperature in Celsius scale using the variable `Cel` and returns the result in line-14. The result is finally displayed in line-9 in the `main()` function.

Factorial of a number using normal function

```
1  /*Program-87: Function to calculate Factorial of a number*/
2  #include<stdio.h>
3  long int Factorial(int n);
4  void main()
5    { int x; long int f;
6      printf("\nEnter number to find factorial: ");
7      scanf("%d", &x);
8      f = Factorial(x);
9      printf("\nRequired factorial is %ld", f);
10   }
11 long int Factorial(int n)
12   { int count=1;
13     long int fact=1;
14     while(count<=n)
15        {fact=fact*count;
16         count++;
17        }
18     return fact;
19   }
```

Output

```
Enter number to find factorial:  4
Required factorial is 24
```

In the above program, the **main()** function inputs the number whose factorial is to be found out in line-7 and calls the factorial function in line-8 with the argument **x**. The function receives the value stored in **x** within the argument **n**. The variable **fact**, declared as a long integer in line-13 within the **Factorial()** function, is used to store the result of the calculation. The factorial is calculated using the **while** loop of line-14 and the result returned to the **main()** function in line-18. Within the **main()** function the required result is displayed in line-9.

Calculating nPr and nCr using function

```
1  /*Program-88: Calculation of nPr and nCr using factorial function*/
2  #include<stdio.h>
3  long int Factorial(int x);
4  void main()
5    { int n, r; long int nPr, nCr;
6      printf("\nEnter value of n: ");
7      scanf("%d", &n);
8      printf("\nEnter value of r: ");
9      scanf("%d", &r);
10     nPr = Factorial(n)/Factorial(r);
11     nCr = Factorial(n)/(Factorial(r)*Factorial(n-r));
12     printf("\nRequired nPr = %ld, nCr = %ld", nPr, nCr);
13   }
14 long int Factorial(int x)
15   { int count=1;
16     long int fact=1;
17     while(count<=x)
18       {fact=fact*count;
19        count++;
20       }
21     return fact;
22 }
```

Output

```
Enter value of n:  5
Enter value of r:  2
Required nPr = 60, nCr = 20
```

The above program is used to calculate the values of ${}^{n}P_{r}$ and ${}^{n}C_{r}$. It inputs the values of **n** and **r** in lines-7 and 9 respectively of the **main()** function.

In line-10, the **Factorial()** function is called twice – once with the value of **n** and then with the value of **r**. The **Factorial()** function defined in line-14, is used to calculate both these factorials. Line-11 of the **main()** function then calls the **Factorial()** function thrice – with the arguments (**n**), (**r**), and (**n-r**). In the first case the actual argument **n** is received by the formal argument **x** in the **Factorial()** function. The value of **r** is received by **x** during the next function call. Finally in the third case the value of (**n-r**) is received by **x** during the last function call. Each time, the **Factorial()** function calculates the factorial of the number that it receives within its variable **x** and returns the result via the **return** statement of line-21.



Sum of digits of a number using function

```
1   /*Program-89: To find the sum of the digits of a number using a function*/
2   #include<stdio.h>
3   int SumDigit(int num);
4   void main()
5      { int n, s;
6        printf("\nEnter value of number: ");
7        scanf("%d", &n);
8        s = SumDigit(n);
9        printf("\nRequired sum of the digits = %d", s);
10     }
11  int SumDigit(int num)
12     { int sum=0, digit;
13       while(num>0)
14         {digit = num%10;
16          sum = sum + digit;
17          num = num/10;
18         }
19       return sum;
20     }
```

Output

```
Enter value of number:  2198
Required sum of the digits = 20
```

The above program is used to calculate the sum of the digits of a number using a function. It inputs the values of the number **n** in lines-7 of the **main()** function. Then the function **SumDigit()** is called in line-8 with the argument as **n**. The function starting in line-11 receives the value stored in **n**, within the variable **num**. The **while** loop starting from line-13 then calculates the sum and returns the result stored in the variable **sum** to **main()** function. The result is displayed in line-9.



Checking from a range of numbers for prime

```
1   /*Program-90: To check from a range of numbers, whether a number is prime or not*/
2   #include<stdio.h>
3   void IsPrime(int num);
4   void main()
5   { int start, end, i;
6     printf("\nEnter the starting number of the range of numbers to check if prime: ");
7     scanf("%d", &start);
8     printf("\nEnter the ending number of the range of numbers to check if prime: ");
9     scanf("%d", &end);
10    for(i=start; i<=end; i++)
11       IsPrime(i);
12  }
13  void IsPrime(int num)
14  { int j, flag=1;
15    for(j=2; j<num; j++)
16       if(num%j == 0)
17         { flag=0; break; }
```

```
    if(flag==1)
18      printf("\nThe number %d is prime", num);
19
20  }
```

Output

```
Enter the starting number of the range of numbers to check if prime: 5
Enter the ending number of the range of numbers to check if prime: 10
The number 5 is prime
The number 7 is prime
```

The above program finds **whether a number is prime or not from a range of numbers**. The starting number in the range is entered in line-7 in the variable `start` and the ending number in the range is entered in line-9 in the variable `end`. The `for` loop in line-10 then reads numbers from `start` to `end` and with each number it calls the `IsPrime()` function to check if that nmber is prime or not.

The function `IsPrime()` receives the value in the variable `num`. It then checks if the value in `num` is prime or not using the `for` loop of line-15. A `flag` variable is used to check if the number could be divided without a remainder or not. In case of proper division the flag variable is made equal to 0. For a prime number flag remains at 1. Finally it prints whether the number is prime or not by checking the value of flag. Note that as the function is not returning any value, its return data type is taken as `void`.

The next program shows how the **hcf of two numbers** can be calculated using recursion. We know that the hcf of two numbers can be found out by using the method of dividing one number by the other. If the result of dividing one number by the next gives a '0' remainder, then the divisor is the required hcf. Otherwise the original divisor is made the new dividend, the remainder is made the new divisor and the process repeated till a '0' remainder is found (refer to chapter 11).



HCF (GCD) of numbers using recursion

```
1  /*Program-91: To calculate the hcf/gcd of two numbers using recursion*/
2  #include<stdio.h>
3  int hcf(int x, int y);
4  void main()
5  { int num1, num2, h;
6    printf("\nEnter the first number: ");
7    scanf("%d", &num1);
8    printf("\nEnter the second number: ");
9    scanf("%d", &num2);
10   h = hcf(num1, num2);
11   printf("\nThe required hcf is = %d", h);
12 }
13 int hcf(int x, int y)
14 { if(x%y == 0)
15     return y;
16   else
17     return hcf(y, x%y);
18 }
```


Output

```
Enter the first number: 40
Enter the second number: 50
The required hcf is = 10
```

In the above program, the numbers are entered in lines-7 and 9. Then, the `hcf()` function is called in line-10. Within the `hcf()` function, the value of `num1` is received by `x` and that of `num2` by `y`. Line-14 checks if the remainder of dividing `x` by `y` is '0'. If so, it returns `y` (the divisor) as the required hcf. Otherwise, the `hcf()` function is recursively called in line-17 with the arguments `y` and the remainder `x%y` as the new dividend and divisor.

The next program shows how one can find the value of the nth term of the Fibonacci series using recursion. From the Fibonacci series we know that the first two terms of the series are fixed and equal to '1'. Any other term can be found by adding the previous two terms. Therefore for n=1 and n=2 (where n is the term number) we have the stopping condition, whereby '1' is returned. For any other term 'n', we have Fibonacci of term 'n' i.e. **Fibonacci(n) as Fibonacci(n-1) + Fibonacci(n-2).**



Finding nth term of Fibonacci series using recursion

Fibonacci series: 1, 1, 2, 3, 5, 8, 13, 21, ...

```
1   /*Program-92: To calculate the n-th term of the Fibonacci series using recursion*/
2   #include<stdio.h>
3   long int fibonacci(int n);
4   void main()
5   { int term;
6     long int f;
7     printf("\nEnter the term number: ");
8     scanf("%d", &term);
9     f = fibonacci(term);
10    printf("\nThe required term is = %ld", f);
11  }
12  long int fibonacci(int n)
13  { if(n == 1 || n == 2)
14      return 1;
15    else
16      return (fibonacci(n-1) + fibonacci(n-2));
17  }
```

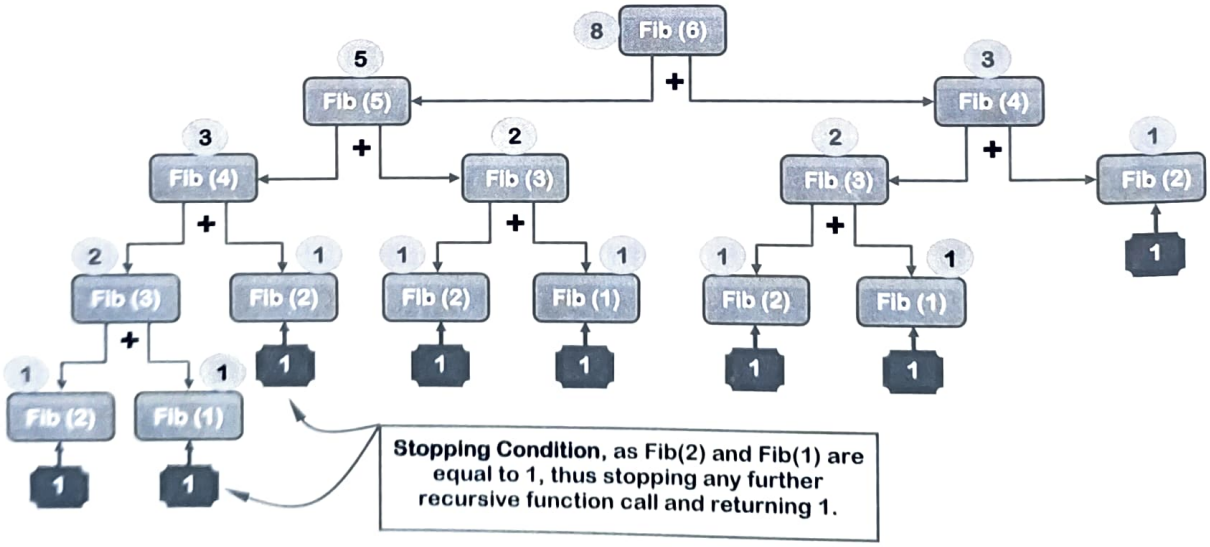
Output

```
Enter the term number: 6
The required term is = 8
```

The **recursion tree** shown below will help the student understand how the recursive calls are made to calculate a particular term in the series. Each function call with term number 'n' is broken down into two function calls with term numbers (n-1) and (n-2), till the stopping condition is reached, when n=2 or 1. At this point the recursive calls stop and the value '1' is returned.



Recursion Tree



The next program uses recursive function call to calculate the **sum of the digits of a number.**



Finding sum of digits of a number using recursion

```
1   /*Program-93: To calculate the sum of the digits of a number using recursion*/
2   #include<stdio.h>
3   int sum(int num);
4   void main()
5   { int n, s;
```

```
6     printf("\nEnter the number: ");
7     scanf("%d", &n);
8     s = sum(n);
9     printf("\nThe required sum is %d", s);
10 }
11 int sum(int num)
12 { int total;
13   if(num == 0)
14      return 0;
15   total = num%10 + sum(num/10);
16   return total;
17 }
```

Output

```
Enter the number: 5768
The required sum is = 26
```

The number whose sum of the digits are to be found, is entered in line-7 and the function **sum()** is called in line-8. In line-15 of the function **sum()**, the rightmost digit of the number is extracted by the expression **num%10** and the function **sum()** is called recursively with the value **num/10**. This recursive function call continues till **num/10** becomes equal to '0'. This is checked in line-13, whereby the value '0' is returned. This condition serves as the stopping condition.

The above function can also be done in a different way as shown below:

```
10 int sum(int num)
11 { if(num < 10)
12      return num;
13   else
14      return (num%10 + sum(num/10));
15 }
```

The following program uses a function to calculate the value of e up to the number of terms entered by the user. The function uses a static variable for the calculation.



Calculating e using static variable

```
1  /*Program-94: Function call having static variable to calculate value of e*/
2  #include <stdio.h>
3  float factorial();
4  int main()
5  {float sum=0;
6   int n, i;
7   printf("\nEnter number of terms to add for e-series: ");
8   scanf("%d", &n);
9   for(i=1; i<=n; i++)
10     sum = sum + 1/factorial();
11  printf("\nThe required value of e is %f:", sum);
12  return 0;
13 }
14 float factorial()
15 {static int count;
16  int k;
17  float fact=1;
18  for(k=1; k<=count; k++)
19      fact = fact * k;
20  count++;
21  return fact;
22 }
```

Output:

```
Enter number of terms to add for e-series: 6
The required value of e is 2.716667
```

The **static** variable **count** declared in line-15 is automatically initialised to **0** when the **factorial()** function is called for the first time. Hence the factorial of **0** i.e. **1** is returned by the function. The **count++** statement of line-20 then increments **count** by **1**. When the function is called for the second time, being a static variable, **count** retains its value of **1** and the **for** loop of line-18 calculates factorial 1. **Count** is next incremented to 2. For the third function call the **for** loop of line-18 therefore calculates factorial of 2. In this way for each function call the last incremented value of **count** is used to calculate the factorial. Within the **main()** function the **for** loop of line-9 then adds the inverse of the factorial values to calculate the value of **e** as e= 1/(0!) + 1/(1!) + 1/(2!) + ... up to **n** terms.

The following program uses a function to **accept two numbers and an operator** and calculates and returns the result as per the operator type.

```
1  /*Program-95: Function call with numbers and operator*/
2  #include <stdio.h>
3  float operation( float x, float y, char optr);
4  int main()
5  {float a, b, answer;
6   char symbol;
7   printf("\nEnter first number: ");
8   scanf("%f", &a);
9   printf("\nEnter second number: ");
10  scanf("%f", &b);
11  fflush(stdin);
12  printf("\nEnter operator: ");
13  scanf("%c", &symbol);
14  answer = operation(a, b, symbol);
15  printf("\nThe required result is %f:", answer);
16  return 0;
17 }
18 float operation( float x, float y, char optr)
19 {float result=0;
20  switch(optr)
21   {case '+': result = x+y;
22             break;
23    case '-': result = x-y;
24             break;
25    case '*': result = x*y;
26             break;
27    case '/': if (y==0)
28                printf("\nDivision by zero error!!!");
29             else
30                result = x/y;
31             break;
32   }
33  return result;
34 }
```

Output:

```
Enter first number: 12.9
Enter second number: 6.7
Enter operator: *
The required result is 86.43
```

The following program uses a function to **convert a decimal number to binary.**



Decimal to Binary conversion

```
1  /*Program-96: Function call to convert from decimal to binary*/
2  #include <stdio.h>
3  void convert(int num);
4  int main()
5  { int n;
6    printf("\nEnter number to convert: ");
7    scanf("%d", &n);
8    convert(n);
9    return 0;
10 }
11 void convert(int num)
12 { long int binary=9;
13   int rem, digit;
14   do{rem = num%2;
15      binary=binary*10+rem;
16      num = num/2;
17     }while(num!=0);
18   printf("\n\nBinary Equivalent\n");
19   while(binary>9)
20     {digit = binary%10;
21      printf("%d", digit);
22      binary = binary/10;
23     }
24 }
```

Output:

```
Enter number to convert: 25
Binary Equivalent
11001
```

The function **convert()** receives the decimal number **num**. The **do-while** loop of line-14 then finds the remainder of dividing the number **num** by 2 (refer to the Data Representation chapter for conversion process). As the final result is obtained by writing the remainders in reverse order, we have to form a number in the forward order from the remainders and then reverse the number so formed to get the result. The variable **binary** is used to store the forward number. As the first remainder can be a 0, the result in line-15 will yield a zero value for **binary** till the first non-zero remainder is obtained. Hence the zero remainders will not get stored. To overcome this problem, in line-12 we have initialised the variable **binary** to a non-zero number (9 in this case. You can take any other non-zero digit) so that the zeroes are retained (9*10+0=90, 90*10+0=900 etc.; Otherwise 0*10+0=0 always for zero remainders, until the first non-zero remainder).

The **while** loop of line-19 is then used to reverse the **binary** number formed. Refer to chapter-11 for the logic behind reversing a number. To eliminate the first 9 in the binary value, the **while** loop of line-19 is run till **binary>9** i.e. till **binary** is a two digit number. The extracted digits are printed in line-12.

The following program uses two functions to print a double cone depending upon the number of lines passed to the function.



Printing pattern using function

```
1  /*Program-97: Using a function to print a double cone*/
2  #include <stdio.h>
3  void pattern1(int n);
4  void pattern2(int n);
5  int main()
6  { int i, j, k, n;
7    Printf("\nEnter number of lines to print for the inverted pyramid: ");
8    scanf("%d", &n);
```

```
9     pattern2(n);
10    pattern1(n);
11    return 0;
12  }
13  void pattern1(int n)
14  { int i, j, k;
15    for (i=1; i<=n; i++)
16       {for (j=1; j<=n-i; j++)
17             printf(" ");
18        for (k=1; k<=2*i-1; k++)
19             printf("*");
20        printf("\n");
21       }
22  }
23  void pattern2(int n)
24  { int i, j, k;
25    for (i=n; i>=1; i--)
26       {for (j=1; j<=n-i; j++)
27             printf(" ");
28        for (k=1; k<=2*i-1; k++)
29             printf("*");
30       printf("\n");
31       }
32  }
```

Output:

```
Enter number of lines to print a double cone: 5
*********
 *******
  *****
   ***
    *
    *
   ***
  *****
 *******
*********
```

The function **pattern1()** prints an upright pyramid and the function **pattern2()** prints an inverted pyramid. The functions are defined in line-13 and 23 respectively. We have already learnt how to print an upright pyramid in chapter-11. The function **pattern1()** uses the same code to print the upright pyramid. The function **pattern2()** is identical to the function **pattern1()** with the exception in line-25 for the outer **for** loop. Since the base is large for an inverted pyramid, the range is taken from **i=n** to **i=1** and the value of **i** is decremented by **i--** for each run of the outer loop.

REMEMBER that for any inverted shape simply change the range of the outermost loop to **i=n** to **i=1**, with the variable **i** decremented by **i--**. All the other loops inside the outer loop will remain exactly the same.

The following program uses several functions to **print the Pascal's Triangle**.



Pascal's Triangle using functions

```
1   /*Program-98: Using function to print Pascal's Triangle*/
2   #include <stdio.h>
3   long int factorial(int num);
4   int nCr(int n, int r);
5   int main()
6   { int i, j, k, n;
7     printf("\nEnter number of lines to print for Pascal's Triangle: ");
8     scanf("%d", &n);
```

```
    for (i=0; i<n; i++)
    {for (j=0; j<n-i-1; j++)
          printf(" ");
     for (k=0; k<=i; k++)
          printf("%d ", nCr(i,k));
     printf("\n");
    }
  return 0;
}
long int factorial(int num)
{long int fact=1;
 int i;
 for(i=1; i<=num; ++i)
   fact = fact*i;
 return fact;
}
int nCr(int n, int r)
{int C;
 C = factorial(n) / ( factorial(r)*factorial(n-r) );
 return C;
}
```

Output:

```
Enter number of lines to print for Pascal's Triangle: 5
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

The above program uses two functions to print the Pascal's triangle. The **for** loop in line-12 within the main code calls the **nCr()** function with the arguments **i** and **k**. The **nCr()** function in turn calls the **factorial()** function three times in line-27 to calculate the value of **nCr**.

In Pascal's triangle the values inside the triangle are formed by adding the top two numbers in the triangle with respect to given nummber.
E.g. 1+1=2, 2+1=3, 1+3=4, 3+3=6

The Pascal's triangle is formed by the coefficients of x in the Binomial expansion of $(1+x)^n$

C to it that .....

The following programs show **some common mistakes** while writing programs using functions.

```
#include<stdio.h>
void main(void)
  { int a, b;
    printf("\nEnter number1: ");
    scanf("%d", &a);
    printf("\nEnter number2: ");
    scanf("%d", &b);
    MaxValue(a,b);
  }
void MaxValue(int x, int y)
 { int z;
   z = (x>y)?(x):(y);
   printf("\nThe maximum of the two numbers = %d", z);
 }
```

When the above program is compiled, it will display the message – "*Function prototype not found*". The reason for this is that the **function is defined after `main()` function is defined**, and **without declaring any function prototype**. To solve this problem, the function prototype can be declared as shown below:

```
#include<stdio.h>
void MaxValue(int x, int y);
void main(void)
  { int a, b;
    Printf("\nEnter number1: ");
    scanf("%d", &a);
```

An **alternative way to declare a function** prototype is as shown below, i.e. within the `main()` function just like a variable declaration.

```
1  #include<stdio.h>
2  void main(void)
3    { int a, b;
4      void MaxValue(int x, int y);
5      printf("\nEnter number1: ");
6      scanf("%d", &a);
7      ... ... ... ... ...
```

The next program shows another common mistake while writing functions:

```
1  #include<stdio.h>
2  int Armstrong(int num);
3  void main()
4    { int x, s;
5      printf("\nEnter value of number: ");
6      scanf("%d", &x);
7      s = Armstrong(x);
8      if(s==1)
9          printf("\n%d is an Armstrong number", x);
10     else
11         printf("\n%d is NOT an Armstrong number", x);
12   }
13 int Armstrong(int num)
14   { int sum=0, num2, digit;
15     num2 = x;
16     while(x>0)
17       {digit = x%10;
18        sum = sum + digit*digit*digit;
19        x = x/10;
20       }
21     if(num2 == sum)
22        return 1;
23     else
24        return 0;
25   }
```

The above program checks whether a number entered by the user is an Armstrong number or not. The value is entered in line 6 into the variable `x`. The program then calls the function `Armstrong()` with the argument `x` to check if it is an Armstrong number.

Within the function, the value of the variable is copied into a second variable `num2`. The `while` loop from line-16 to 20 then adds the cube of the digits of the number and stores the result in `sum`.

The `if` statement in line-21 then compares the value stored in `sum` with the original number copied in the variable `num2`. In case these are same then the function returns '1' in line-22 and in case these are not the same (i.e. the number is not an Armstrong number), the function returns '0' via line-24. Within `main()` that value is received by the variable `s` in line-7. Checking the value of `s` in line-8, the `main()` function prints whether the number `x` is an Armstrong number or not.

However, the above program when compiled will show the message, "*Undefined symbol x*". The reason is that the variable '`x`' declared in line-4 in the `main()` function is active only within the `main()` function as it is an automatic variable. Therefore when the function `Armstrong()` wants to refer to the variable '`x`' in lines 15, 16, 17, and 19, then that variable '`x`' is not available within the function `Armstrong()` and hence shows the message.

When the function `Armstrong(x)` is called in line-7 with the actual argument `x`, the value stored in `x` is copied to the formal argument `num` in line-13. Hence within the function the variable that corresponds to `x` is `num`. **Therefore one has to use `num` in place of `x` within the function.**

The rectified function code is given below:

```
13  int Armstrong(int num)
14    { int sum=0, num2, digit;
15      num2 = num;
16      while(num>0)
17        {digit = num%10;
18         sum = sum + digit*digit*digit;
19         num = num/10;
20        }
21      if(num2 == sum)
22        return 1;
23      else
24        return 0;
25    }
```

The next program is used to check whether a number is odd or even using a function. The value is entered in line-6 and the function **OddEven()** is called in line-7 with the argument **x**. The function then checks if the number is odd or even using the **if** statement of line-14. If the number is odd, it returns '1' else, if it is even it returns '0'. Within **main()** the variable **s** stores this returned value. The **if** statement of line-8 checks **s** and prints if **x** is odd or even.

```
1   #include<stdio.h>
2   int OddEven(int num);
3   void main()
4     { int x, s;
5       printf("\nEnter value of number: ");
6       scanf("%d", &x);
7       s = OddEven(int x);
8       if(s==1)
9           printf("\n%d is Odd", x);
10      else
11          printf("\n%d is Even", x);
12    }
13  int OddEven(int num)
14    { if(num%2 == 1)
15        return 1;
16      else
17        return 0;
18    }
```

However, if the program is compiled, then it will show the error message "*Expression Syntax*". The reason is that **the variable x has been defined twice** – once in line-3 and again in line-7 by writing **OddEven(int x)** during the function call. The rectified **main()** function is shown below:

```
3   void main()
4     { int x, s;
5       printf("\nEnter value of number: ");
6       scanf("%d", &x);
7       s = OddEven(x);
8       if(s==1)
9           printf("\n%d is Odd", x);
10      else
11          printf("\n%d is Even", x);
12    }
```

Therefore always keep in mind that **during a function call only the variable names should be included** as function argument and not their data types along with them.

The program shown below is used to find the greater of two numbers using a function called **maximum()**.

```
1  #include<stdio.h>
2  int maximum(int x, int y);
3  main()
4  {int a, b, max;
5   printf("\nEnter number1: ");
6   scanf("%d", &a);
7   printf("\nEnter number2: ");
8   scanf("%d", &b);
9   maximum(a, b, max);
10  printf("\nThe greater of the two numbers is %d", max);
11 }
12 int maximum(int x, int y)
13  {int m;
14   m = (x>=y)?(x):(y);
15   return m;
16  }
```

The above program will however not compile due to several reasons. First in line-9, when the function is called, we are passing three arguments to the function viz. **a**, **b**, and **max**. The function requires only the variables **a** and **b** to calculate the maximum among these variables. There is no need to send the variable **max** as a function argument along with **a** and **b**. We need **max** to store the result in the **main()** function and not pass it to the function **maximum()** as a function argument.

Therefore one should **pass only those values as function arguments which are required by the function to carry out the calculations**. Any other variable which may be required by the function to carry out any calculation within that function should be declared and used within that function only.

Moreover the function header has been defined in line-2 and 12 to contain only two integer type variables i.e. **x** and **y**, where **x** will receive the value of **a** and **y** will receive the value of **b**. therefore the function header will also not accept the third variable **max**.

The **rectified program** is shown below:

```
1  /#include<stdio.h>
2  int maximum(int x, int y);
3  main()
4  {int a, b, max;
5   printf("\nEnter number1: ");
6   scanf("%d", &a);
7   printf("\nEnter number2: ");
8   scanf("%d", &b);
9   max = maximum(a, b);
10  printf("\nThe greater of the two numbers is %d", max);
11 }
12 int maximum(int x, int y)
13  {int m;
14   m = (x>=y)?(x):(y);
15   return m;
16  }
```



The Fact File

- **A function is a block of instructions that together can perform a certain specific task**
- A C program in general consists of a number of such functions or modules of code; main() being the most important function
- **Library functions are pre-written and pre-compiled functions and stored in the standard library of C and can be used by programmers**

- Examples of library functions are printf(), scanf(), puts(), sqrt(), malloc(), strlen(), strcpy(), etc.
- **User defined functions are written by users for doing specific tailor made jobs as per the user requirement. To use these functions, the user has to first define the function i.e. type the code for the function and then call the function to carry out its specific tasks**
- Any repetitive job that needs to be executed at several points of code can be assigned to a particular function
- **If a program is compartmentalised using functions to carry out different jobs, it becomes easier to follow the program logic as one has to concentrate on a certain portion of the code only**
- Increase in readability by using a function helps in pin-pointing possible errors within a section of the code and accordingly helps in debugging
- **A function carrying out a certain particular work can be reused in more than one program by making libraries of such user-defined functions and calling them in the programs as and when required**
- The request to a function to execute its code is referred to as calling the function
- **If the compiler encounters a function call before it has been declared or defined, then it causes a program error and displays the message that "function prototype not found".**
- Declaring a prototype results in informing the compiler in advance during compilation that this is the function name that will be used by the main() function or other functions and these are the argument list and return data type of the function defined later on
- **A function definition within another function is strictly forbidden**
- A function can be declared within the main() function along with other variable declarations. However, remember that a function can never be defined within another function
- **The function Header consists of the function return data type, the name of the function, and the function argument or parameter list. E.g. float average (int x, int y)**
- The 'return data type' of a function indicates the data type (i.e. int, float, char etc.) of the value that the function sends back to the point from where it was called
- **When a function is called to do some calculations, it may be supplied with some values on which it is supposed to do the calculations. This list of values forms the argument list or parameter list of a function**
- The function Body consists of the code that the function executes to carry out its specific job. It starts after the function header and is enclosed within a pair of curly braces
- **The variables that are passed from the calling function to the called function are called the actual arguments**
- The variables that receive the values in the function header are called formal arguments
- **The number, type and order of formal arguments must match the number, type and order of the actual arguments, for the function to work properly**
- The return statement is used to send back control to the calling function. It also returns any value (can be result of any calculation) that is placed after the keyword return
- **In general a function can contain as many return statements as required, but can return only one value at a time, to the calling function. E.g. void maximum (int x, int y);**
- It is not necessary that the return statement should be placed at the end of the function, it can be placed anywhere in the function
- **Whenever a program branching is there and the output depends on which program segment the program has branched to, then the different branches can each contain a return statement to return a particular value**
- It is not necessary that a function should always return a value
- **To make the compiler understand when a function does not return a value, the keyword void is placed before the function name in the function header. In such a situation, no return statement should be used in the function**
- In C a function can be made to call itself and a function that has been constructed in such a manner as to call itself is known as a recursive function
- **For a function to be recursive, it must satisfy two basic conditions:**
  It must have an ending point or stopping condition
  It must make the problem simpler
- In C every variable has two attributes i.e. the scope and the class of the variable
- **By scope of a variable we mean the area of the program in which the variable is valid**

- By class of a variable we mean whether the variable is temporary or permanent
- **If the storage class of the variable is not mentioned in the declaration, then the compiler assigns the default storage classes based on the context in which the variable was used**
- The variable's storage class determines in which location of the computer, the values in the variables are stored i.e. whether in the Memory or in the CPU Registers (which are special memories located within the CPU)
- **Based on the scope and class of the variables, these are divided into 4 different types. These are:**
  - **Automatic or local storage class variables**
  - **Register storage class variables**
  - **Static storage class variables**
  - **External or Global storage class variables**
- The current value of an automatic variable is the value of that particular variable in the current block
- **Register Storage Class variables are similar to auto variables with the only exception that they generally occupy the CPU register memory spaces instead of the conventional memory**
- Static Storage Class variables also have scope within the block in which they are declared, but they are initialised only once and their default initial value is '0'
- **In case of static variables, once initialised, the value persists even between different function calls**
- External Storage Class variables differ from other types mainly with respect to their scope. They have a global scope i.e. they are not defined or available within a particular block of code but are available throughout the program

Review Questions

Q1. **Multiple Choice Questions. Select from any one of the four options.** **1 each**

i) A function in C cannot:
a. return a single value only
b. have multiple return statements
c. be defined within another function
d. have a void return type

ii) Which of the following is not a part of a function header in C?
a. function name
b. function parameter list
c. function return data type
d. function local variables

iii) A function:
a. must return a value
b. may return a value
c. can return multiple values
d. always returns an integer value

iv) A function can return how many values?
a. 1 b. 2 c. 3 d. none of these

v) A function header:
a. ends with a semicolon
b. cannot contain any float data as arguments
c. by default has an int return data type
d. cannot have a char return data type

vi) Pre-written and precompiled functions available for writing a C program are generally called:
a. import functions b. stored functions c. predefined functions d. library functions

vii) For a recursive function which of the following statements is not true:
a. must have a stopping condition
b. can be expressed in terms of itself
c. uses a stack for its working
d. can be always used as a replacement for a loop

viii) Which of the following does not represent a storage class keyword in C?
a. external b. auto c. static d. register

ix) Which of the following relations cannot be expressed using a recursive function?
a. Q(y) = Q(y-1)*3+y and Q(0) = 5
b. P(a, b) = P(a-1, b-2) + (a-b) and P(1,1) = P(-1,5)
c. S(p,q) = p*S(p-1,q) and S(0,q) = 1
d. T(n) = (n-1)*T(n-1) and T(1)=10

x) Arguments passed to a function are generally called:
a. real arguments b. actual arguments c. formal arguments d. passed arguments

xi) Arguments received by a function are generally called:
a. formal arguments b. copied arguments c. actual arguments d. received arguments

xii) Which of the following keywords is used to indicate that a function does not return anything?
a. null b. not c. none d. void

xiii) Which storage class can be used to keep a function variable active between function calls?
a. register b. static c. automatic d. extern

xiv) Which storage class can be used to access a variable globally?
a. register b. automatic c. extern d. static

xv) Which storage class is normally the default storage class of a variable?
a. extern b. static c. register d. automatic

xvi) The order in which the function arguments are passed and the order in which these are received:
a. does not matter b. has to be same c. may or may not be same d. sometimes matter

xvii) A function prototype:
a. must be included before the function is defined
b. may or may not be included when a function is defined
c. needs to be included if the function is defined after the function call
d. is compulsory to use

xviii) For a function taking no arguments:
a. can return only an integer value
b. can have actual arguments during function call
c. cannot return a value
d. will not have any formal arguments

xix) What is the return value of the C function given below, if the formal arguments w=5 and x=5.5?
Fn (int w, float x) { return w+x; }
a. 10.5 b. 5.5 c. 10 d. 5

xx) What is the return value of the C function given below, if the formal arguments a=4 and b=3?
Fn (int a, int b) { return (float)(b+a)/2; }
a. 4 b. 3.5 c. 7 d. 3

xxi) What is the return value of the C function given below, if the formal argument x=3?
int Fn (int x) { if (x=2) return x; else return 1; }
a. 1 b. 2 c. 3 d. none of these

xxii) What is the return value of the C function given below, if the formal argument x=5?
```
float Fn (int x)
  {if (x==3) return 1;
   return x/Fn(x-1); }
```
a. 1 b. 1.25 c. 2 d. none of these

Q2. **Short Answer type questions:** **1 each**



i) State one utility of using functions in a program.
ii) State the three parts of a function header.
iii) What do you mean by actual argument?
iv) What do you mean by formal argument?
v) What is the use of the return keyword in a function in C?
vi) What is a function prototype?
vii) Where do you declare a function prototype?
viii) A function can return at the most how many values?
ix) What keyword is used to indicate that a function is not returning anything?
x) Name any one storage class in C.
xi) Which is the default storage class of a local variable in C?
xii) State any one condition that is to be satisfied by a function to be recursive.

Q3. **Long Answer type questions:** **7 each**



i) Write a function to calculate and return the factorial of a number passed to it as argument. Explain the difference between actual and formal arguments in C. What is the default return data type of a function? 4+2+1

ii) Write a function to check if a number passed to it is prime or not. For prime, the function returns 1 and for non-prime, it returns 0. What do you mean by function recursion? What is the purpose of the return statement in a function? 4+2+1

iii) With the help of a code example show how a function can have more than one return statement in it. Name the different types of storage classes in C. What keyword is used to indicate that a function does not return anything? 4+2+1

iv) Give an example for a function that receives arguments, but does not return anything. Variables of which storage class remain active between function calls? What are the conditions that need to be satisfied for a function to be recursively expressed? 4+1+2



Q4. **Assignment Programs:** **4 each**

i) Write a program in C using a function called ChangeCase( ) that receives a character and checks if the character is in lowercase or in uppercase. If the character is in lowercase then it converts it to uppercase and if it is in uppercase then it converts it to lowercase and returns the character to the main( ) function where the changed character is displayed along with the original character.

ii) Write a program in C using a function called CharType( ) that receives a character and displays "An Alphabet" if the character is an alphabet, "A Digit" if the character is a digit from 0 to 9 or "Other Character" if the character is neither an alphabet nor a digit. The function does not return any value and displays the result within the function.

iii) Write a program in C using a function called floor( ) to take any floating point number and convert it to the next lowest integer and return the result as a float. For example:

- For input value 4.95 the return value will be 4.0
- For input value 7.0 the return value will be 7.0
- For input value – 2.5 the return value will be –3.0 etc.

Hint: Use an integer variable in the floor( ) function and assign the float variable to round off, to this integer variable. Reassign the integer variable to the float variable to convert the data type again from integer to float. Return this value to the main( ) function.

iv) Rewrite the floor( ) function created in the last program and use it in three other functions called Round_to_Int( ), Round_to_Tenths( ), and Round_to_Hundredths( ) to round off a float number to the nearest integer, up to a single decimal place and up to two decimal places respectively.

E.g. the relation y = floor(x+0.5) rounds x up to the next higher integer.
y = floor(x*10+0.5)/10 rounds x up to the tenth's position.
y = floor(x*100+0.5)/100 rounds x up to the hundredth's position

v) Write a program in C which uses a function to check whether a number passed to the function is a prime number or not. The function prints the number if prime. Now use this function in the main function to find all prime numbers between any two numbers x and y entered by the user (y>x).

vi) Write a function that takes two integer values and returns the hcf (or gcd) of the two numbers. Within the main function input n integer values using a loop and find the hcf of all the input numbers by properly using the defined hcf function (hint: find the hcf of the first two numbers and then get a new hcf by using the current hcf and a new number entered).

vii) Write a C program that has the following functions: sin(x), cos(x), cosec(x), sec(x), tan(x) and cot(x). Write the functions for sin(x) and cos(x) using the series representation of sine and cosine and taking up to 10 terms for each series. The remaining functions should use the sine(x) and cos(x) functions as per need to get their results. Within the main() code enter an angle in radians and use a switch case construct to ask the user for the type of function to use and get the result accordingly. Example of the functions: ***float sine(float x)***

viii) Write a program in C to find the sum of the digits of a number using a recursive function.

ix) Write a program in C to find the sum of the series given below using a recursive function.
S = 3 + 6 + 9 + 12 + 15 + 18 + ... + 3n (i.e. up to n terms)
The user enters the number of terms to add i.e. n in the main( ) function.

x) Write a program which uses a recursive function call to find the nth Lengendre Polynomial given recursively by:

- $P_0 = 1$
- $P_1 = x$
- $P_n = \{ (2n - 1)/n \}*x*P_{n-1} - \{ (n - 1)/n \}*P_{n-2}$

Where x is any floating point between –1 and +1 i.e. -1<x<+1, and n = 2, 3, 4, ... The values of x and n are input by the user (hint: Pass both n and x to the recursive function).

CHAPTER 13
Arrays in C

- Introduction to Arrays 13-1
- How to declare an Array 13-1
- How to enter and read data from an Array 13-3
- Multi Dimension Arrays and Matrices 13-5
- Arrays and Pointers 13-9
- Some worked out examples 13-15

13.1 Introduction to Arrays

Till now we have written programs where we worked with a limited number of integers, floats or character type data. However consider the situation of storing the marks secured by 100 students in a class for the subject Physics. With the knowledge that we have so far, for storing the marks we need to declare 100 integer type variables like phy001, phy002, phy003 ... phy100. However this is an inconvenient proposition, as it will be very difficult to deal with 100 such variables at a time, for example to get the average of the marks scored. Now if we consider six different subjects in a particular class and 100 students in a class, then we have to deal with 6x100=600 different variables! This is surely not the right way to handle such large number of values and there should be a proper way out.

An array is a collection of similar type of variables under a common variable name.

To store similar types of data for a common purpose, instead of declaring a large number of variables for a large number of data we use a special type of storage mechanism called an array. **An array is a collection of similar type of variables under a common variable heading** or name. It serves as a container or Data Structure to store a collection of data. Thus arrays can be a collection of marks obtained by the different students in a class, the individual age of all employees of a company, it can be the daily average temperature readings of a city, it can be a collection of first names or last names of students in a school etc.

What is an array

As data can be of various types like integers, floats, characters etc. accordingly can have a collection or **Array of integers, floats, characters** etc. But remember that **all elements of any given array must be of the same data type**, i.e. we cannot have an array of numbers some of which are `int` and some `float` etc. Either all should be `int` or all should be `float` or all should be `char` etc. The different values stored in the array are called **Array Elements**.

All the elements in an array must be of the same data type.

13.2 How to declare an Array

Like all other variables, an array needs to be declared first before it can be used. The declaration consists of stating the type of data used in the array, the name of the array and the number of elements stored in the array as shown below:

```
int physics[20];
```

All the elements in an array occupy consecutive memory locations

In the above declaration, the set of square brackets **[ ]** tells the compiler that we are not dealing with a simple variable but an array. The above example declares an array named `physics` (just like other identifier or variable names) which can store `20` `int` type values.

```
float AvgTemp[365];
```

Similarly the above statement declares an array to store the daily average temperature for a city over a year (i.e. 365 days).

In general, suppose we have a finite series **S** of 7 numbers as shown below:

S = 2, 5, 8, 11, 14, 17, 20

Then the 1st element of the series is 2, the 2nd element is 5, 3rd element is 8 ... up to the 7th being equal to 20. Now if we declare an array called **S** to hold these 7 numbers, then the declaration portion of the array will look like this:

```
int S[7];
```

- Data type of each value in the array is an int
- Name of the array is S
- The array stores 7 values

Declaring an array

How array elements are stored in memory

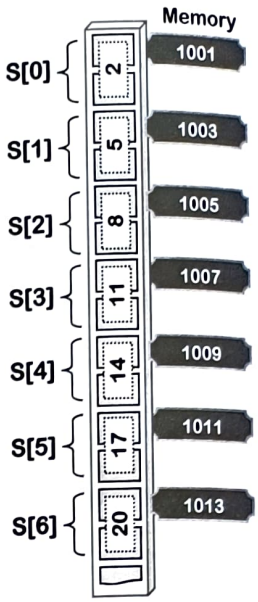
Let us now find how to denote the different **array elements** represented by the array `int s[7]` in C.

The **first** element of the array s is → `s[0]=2` (**NOT** `s[1]`)

The **second** element of the array s is → `s[1]=5` (**NOT** `s[2]`)

The **third** element of the array s is → `s[2]=8` (**NOT** `s[3]`)

The **seventh** element of the array s is → `s[6]=20` (**NOT** `s[7]`)



An array index starts from 0 (zero). For an n element array the index goes from 0 to n-1

Thus the **numbering of array elements starts from 0 and not from 1**, i.e. an array with 100 elements will have the elements numbered from '0' to '99' and not from '1' to '100'. In general `array[n]` will have data elements from `array[0]` to `array[n-1]`.

When one declares an array, C sets aside sufficient memory to store the array elements, depending on the **type of data** and the **number of elements declared**. The array elements are stored in **consecutive memory locations** depending upon the type of data.

For example for the above array `s[7]` declared with 7 integer values, if C assigns the memory location 1001 (say) to the first element, then we will have `s[0]` stored in location 1001, `s[1]` stored in memory location 1003 and so on. Note that, as an `int` type data occupies 2 bytes therefore two bytes of memory i.e. 1001 and 1002 are allotted for `s[0]`, similarly 2 bytes for `s[1]` etc. Hence for the `int` type array the consecutive memory locations are filled up as 1001, 1003, 1005 etc. as shown in the diagram above (for a 4 byte integer, the address of `s[1]` will be 1005 and so on).

Remember that while **declaring an array, the total number of elements needs to be declared.** However, while **accessing the elements**, the index starts **from 0 to one less** than the total number of elements i.e. from 0 to (7-1) = 6, for an array with the above 7 elements. This point should be carefully noted as this may result in common future bugs. The following section will clarify the point.

The fact that the indexing starts from '0' may be thought of as the **offset number** of an array element. The array index gives the **position** of that array element **with respect to the first array element**. Hence `array[1]` indicates an element 1 unit farther from the first element, `array[15]` indicates an element 15 units away from the first element etc. The interpretation will be clear when later we learn about pointers.

Entering data into an array

13.3 How to enter and read data from an Array:

There are two methods to enter data into an array. These are by initialising an array with data elements during design time or by entering data from the keyboard during run time. First let us see how to **enter data from the keyboard**. The following program shows the method.

Normally a for loop is used to access an array.

```
1  /*Program-99: To find the average of a set of numbers using an array*/
2  #include<stdio.h>
3  int main()
4  {int physics[20], i, sum=0;
5   float average;
6   for(i=0; i<20; i++)
7     {printf("Enter marks for student-%d: ", i);
8      scanf("%d", &physics[i]);
9      sum = sum + physics[i];
10    }
11  average = sum/20.0;
12  printf("\nThe average in Physics for 20 students = %.2f", average);
13  return 0;
14 }
```

Output:

```
Enter marks for student-0: 76
Enter marks for student-1: 80
Enter marks for student-2: 66
... ... ...
Enter marks for student-19: 88
The average in Physics for 20 students = 74.32
```

The first thing to consider while inputting data for an array is that we have to repeatedly call the array and insert each of the values for the array elements. This repetitive process calls for using a loop to enter the data. The **for**-loop is used in general to input data into an array (though the **while** and **do-while** loops can also be used if required). The above example enters the marks of 20 students in physics into the integer array named **physics[]** which is declared in line-4. The program then calculates the average of the marks.

The **for** loop in the program starts from line-6. Note that the value of **i** starts from '0' and not from '1' as the first element of the array is **physics[0]** and NOT **physics[1]**. When **i=0**, in line-8 the **scanf()** function enters the value in the position **physics[0]**. In line-9 the number in **physics[0]** is added to the variable **sum**. After that the counter **i** is incremented to '1' by the **i++** operation. In the next iteration in line-8 the **scanf()** function enters the value into the variable **physics[1]**. This value is then again added to the variable **sum** in line-89.

In this manner the **for** loop in the above program repeats 20 times (for **i=0** to **19**) to input the 20 individual marks into the array elements **physics[0]** to **physics[19]**. We have used the same loop to read data and then to do the **sum** of the values in line-9. In line-11, the average is calculated and stored in the variable **average**. It is subsequently printed in line-12.

The next program is used to calculate the standard deviation of a set of numbers. The standard deviation is given by the relation S.D. $= \sqrt{\frac{1}{n}\sum_{i=1}^{i=n}(X_i - \bar{X})^2}$, where $\bar{X}$ denotes the average of a total of **n** values and X_i denotes the individual i^{th} value.

Here the **average needs to be calculated first**, before calculating the standard deviation. As before the average is calculated using the input loop, and a second loop is used to calculate the standard deviation. The **<math.h>** header file is being included to calculate the square root using the **sqrt()** function.



Standard deviation using an array

```
1  /*Program-100: To find the standard deviation of a set of numbers using an array*/
2  #include<stdio.h>
3  #include<math.h>
4  #define MAX 20
5  int main()
6  {float physics[MAX], sum=0, average, sd1=0, sd2;
7   int i;
8   for(i=0; i<MAX; i++)
9     {printf("Enter marks for student-%d: ", i);
10     scanf("%f", &physics[i]);
11     sum = sum + physics[i];
12    }
13  average = sum/MAX;
14  for(i=0; i<MAX; i++)
15    sd1 = sd1 + (average - physics[i])*(average - physics[i]);
16  sd2 = sqrt(sd1/MAX);
17  printf("\nThe required standard deviation in Physics marks = %.2f", sd2);
18  return 0;
19 }
```

Output:

```
Enter marks for student-0: 76
Enter marks for student-1: 80
Enter marks for student-2: 66
... ... ...
Enter marks for student-19: 88
The required standard deviation in Physics marks = 2.32
```

In line-4 we have defined a macro **MAX** to represent the value **20**. In line-6 we have declared a **float** type array called **physics[]** to store **MAX** (i.e. 20) number of **float** type values. The **for** loop in line-8 is used to enter the values into the **physics[]** array as in the previous program. The **sum** is calculated in line-11. The **average** is calculated in line-13 by dividing the sum by the total number of elements i.e. **MAX**.

The **for** loop in line-14 is then used to calculate the sum of the squares i.e. Σ**(average − x[i])²**. The expression **(average-physics[i])*(average-physics[i])** in line-15 is used to calculate the square. The required sum is stored in the variable **sd1**.

Finally in line-16 the required standard deviation is calculated by taking the square root of the expression **sd1/MAX** (as **MAX** indicates the number of elements n) using the **sqrt()** function and storing the result in the variable **sd2**. The calculated standard deviation is printed in line-17.

While entering data into an array, **care should be taken** so that **the input data does not surpass the number of elements declared**. This means that if a 10-element array is declared, then the loop should be used to enter 10 elements only. Otherwise the 11th and subsequent elements can be written onto memory locations reserved for other data, leading to unpredictable results. The compiler will not show any errors and hence this may remain unnoticed initially.



Initialising an Array

Initialising an Array:

Next we discuss how to **initialise an array, which is our second method of entering data** into an array. Just like initialising any other variable, we initialise an array by writing down the initial data elements that the array can hold. (An un-initialised array contains garbage data, unless it has been declared as a static variable array). The method to do this is to **write down the array elements separated by commas within curly brackets**, as shown below.

An array can be initialised only during declaring the array.

```
int weight[7] = {100, 50, 20, 10, 5, 2, 1};
int weight[] = {100, 50, 20, 10, 5, 2, 1};
float constants[2] = {3.141593, 2.718278};
```

The first example shows a 7-element array called **weight[7]**, declared as containing **int** type data. The 7 integer values placed after = within a pair of curly braces { } in the declaration indicates the 7 elements in the array. Thus the **following values are automatically assigned** to the respective array positions:

```
weight[0] = 100
weight[1] = 50
weight[2] = 20
weight[3] = 10
weight[4] = 5
weight[5] = 2
weight[6] = 1
```

Similarly the third array **constants[2]** indicates a 2-element array of floating point values, where:

```
constants[0] = 3.141593
constants[1] = 2.718278
```

The **second example** shows an array whose element number is not declared inside the square brackets. By default **an array assumes the number of elements as indicated by the number of initial values,** which in this case are 7. Hence examples 1 and 2 are the same in effect. However, in case the number of elements of an array is specified within square brackets but all the elements are not initialised, then **the un-initialised elements will all be initialised to 0 automatically**.

Thus for the following example:

```
int fibo[5] = { 1, 1, 2 };
```

The following values will be assigned to the array elements:

```
fibo[0]=1
fibo[1]=1
fibo[2]=2
fibo[3]=0
fibo[4]=0
```

Note that `fibo[3]` and `fibo[4]` are **automatically initialised to 0**.

In case you need **to initialise the whole array to '0'**, you can simply write:

```
int fibo[10] = {0};
```

All the ten elements in the array `fibo[10]` will be initialised to 0. Array initialisation is needed in case we use operations like `A[i]=A[i]+k;` or `B[i]=B[i]*k;` where k is any term.

Initialising an array with '0'

13.4 Multi Dimension Arrays and Matrices

By multi-dimension arrays we mean arrays with **more than one set of elements** and are of the form `A[m][n][p]..[t]` where m, n, p, ... t are the different indexes and each `[]` bracket represents one dimension. The first of a multi-dimension array is a **2 dimensional array also called a matrix** with ***m*** rows and ***n*** columns. The numbering of each row starts from '0' and extends up to one less than the total number of rows. Similarly the numbering of each column starts from '0' and extends up to one less than the total number of columns. The combination of a row and a column number gives the position of any value in the matrix.

Representing a matrix

A 2 dimensional array is called a matrix.

Row →				
	0	0	0	0
	1	1	1	1
	2	2	2	2

Column →				
	0	1	2	3
	0	1	2	3
	0	1	2	3

Row ↓ / Column →				
	0,0	0,1	0,2	0,3
	1,0	1,1	1,2	1,3
	2,0	2,1	2,2	2,3

The figure on the right shows the naming of a 3 by 4 matrix. The rows are numbered as 0, 1, 2 (total 3 rows) while the columns are numbered as 0, 1, 2, 3 (total 4 columns). The **row number and column number together identify any element** within the matrix. If the 3x4 matrix is named as M, then the element in **row no. 2** and **column no. 1** can be written as M_{21} or **M[2][1]** (and **not** M_{32} or M[3][2]).

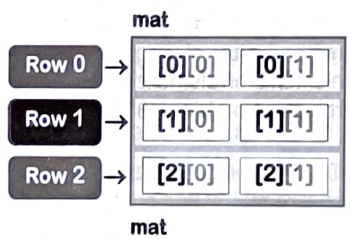
Just like an array, a matrix can either be initialised or the values can be input. Let us now see how to initialise a matrix. It will be easier to treat a matrix of **m** rows and **n** columns as an array with **m** elements, each of which in turn is an array with **n** elements. With this approach let us see how to initialise a 3x2 matrix `int mat[3][2]` (three rows and two columns).

```
int mat[3][2] = { {elements of row-0}, {elements of row-1}, {elements of row-2}};
Or int mat[3][2] = { {110, 220}, {111, 221}, {112, 222} };
Or int mat[3][2] = { {110, 220},
                     {111, 221},
                     {112, 222}
                   };
```

3 rows

2 columns

Matrix initialisation



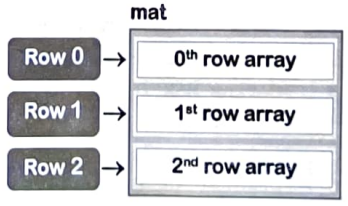
The first index of a matrix indicates the row number and the second index the column number.

The above declaration shows how matrix can be initialised using the concepts of an array. By properly aligning the initial values we can clearly see the 3x2 nature of the array. This concept can be further extended to initialise 3 dimensional arrays. A 3D array can be conceived as **a set of 2D matrices**. Thus a 3x2x4 array can be thought to consist of **3 matrices**, each table having **2 rows** and **4 columns**.

You can altogether do away with the brackets by writing all the elements in a single line separated by commas as shown below.

```
int mat[3][2] = {110, 220, 111, 221, 112, 222};
```

In the above representation, the **compiler will automatically separate** the above elements into 3 rows and 2 columns.



In this respect note that while initialising a matrix, the **row index is optional but the column index must be given**. This is essential because as said earlier, the elements are arranged linearly in memory from lower memory location to higher memory location and the compiler chops the continuous chain of elements based on the number of columns present, to separate the elements into different rows. Hence the initialisation shown below is perfectly valid and the same as the one shown in the previous page.

```
int mat[][2] = {110, 220, 111, 221, 112, 222};
```

To initialise all elements of a matrix to '0', we have to simply write the following code:

```
int mat[3][2]={0};
```

All the elements of an array can be initialised to zeroes by a code like:

Int arr[MAX]={0};

Important Note: One thing should be noted while initialising arrays. In **K&R** C, only external data type arrays (i.e. arrays declared outside all functions and having a global scope) can be initialised. To initialise an array *within* any function, you have to declare an array as `static`. However **ANSI** C permits the initialisation of arrays locally. Hence all compilers compatible with **ANSI** C will allow initialisation of automatic/local arrays. However to be on the safe side one can always declare a local array as `static` as shown below to initialise it locally irrespective of the type of compiler used.

```
int main()
 { static int array[10]={2, 6, 8, 4, 3, 7, 9, 5, 1, 4};
   ... ... ...
 }
```

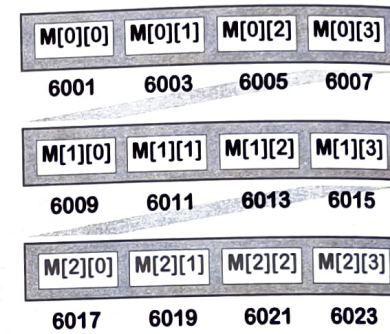


Multi-dimensional Arrays

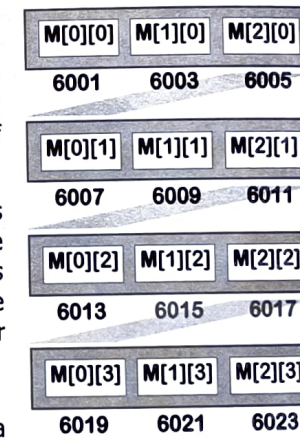
How multi-dimension Array elements are stored in the Memory:

A Matrix is stored in memory either as a row major or as a column major matrix.

Just like a one-dimensional array where the array elements are stored consecutively, the elements in a matrix are also stored linearly in a consecutive manner. In a matrix, the rows can be thought of as laid out consecutively one after the other in the memory such that, where the first row ends the second row begins and so on and so forth. The diagram on the right illustrates the fact for a matrix `M[3][4]`. The memory allotment starts from location 6001 and increases in steps of 2 until the element `M[0][3]` is reached. After that the next memory location 6009 is allotted to the element `M[1][0]`. The **general rule** is that arrays are stored in memory such that **the rightmost index value varies most rapidly**.



It is also evident from the above diagram as to **why the second index i.e. the column index in a two dimensional array is essential**. While going through the memory locations, a new row is indicated after the memory space proportional to the number of columns have been traversed. Thus the way the elements are stored takes a new row automatically after the number of columns have been traversed.



In a Row Major matrix the rows are stored such that when one row ends the elements of the next row begin.

The above method of storing the elements in memory is known as **row major** configuration. Another way of storing the values is the **column major** form, where the elements are stored column wise. In this method after storing the elements of a column in consecutive locations, the elements of the next column is stored as shown in the diagram on the right for the same matrix `M[3][4]`.

In a Column Major matrix the columns are stored such that when one column ends the elements of the next column begin.

The following program shows the use of matrices. The program declares a square matrix and prints the sum of each row separately.

```
1  /*Program-101: To display the sum of the rows of a square matrix*/
2  #include<stdio.h>
3  #define MAX 10
4  int main()
5  {int mat[MAX][MAX], i, j, sum, n;
6   printf("\nEnter total number of rows or columns in the matrix: ");
7   scanf("%d", &n);
```

```
8   for(i=0; i<n; i++)
9     for(j=0; j<n; j++)
10      {printf("\nEnter element-[%d][%d]: ", i, j);
11       scanf("%d", &mat[i][j]);
12      }
13  printf("\nThe input matrix is:\n");
14  for(i=0; i<n; i++)
15    {for(j=0; j<n; j++)
16      {printf("%d\t", mat[i][j]);}
17     printf("\n");
18    }
19  for(i=0; i<n; i++)
20    {sum=0;
21     for(j=0; j<n; j++)
22         {sum = sum + mat[i][j];}
23     printf("\nThe sum of row[%d] is: %d", i, sum);
24    }
25  return 0;
26  }
```



Printing as a matrix

Output:

```
Enter total number of rows or columns in the matrix: 3
Enter element-[0][0]: 2
Enter element-[0][1]: 4
Enter element-[0][2]: 1
Enter element-[1][0]: 7
Enter element-[1][1]: 4
Enter element-[1][2]: 5
Enter element-[2][0]: 2
Enter element-[2][1]: 1
Enter element-[2][2]: 7
The input matrix is:
2       4       1
7       4       5
2       1       7
The sum of row[0] is: 7
The sum of row[1] is: 16
The sum of row[2] is: 10
```

A matrix called **mat** is declared in line-5 containing **MAX** (i.e. 10) number of rows and columns. As we are dealing with a square matrix, the number of rows is equal to the number of columns. The actual number of rows or columns required by the user is input in the variable **n** in line-7.

The nested **for** loop of lines-8 to 12 are used to input the values in the matrix. For **n=3** (as shown in the output), the outer **for** loop of line-8 will run for i=0 to i=2. For every new value of **i**, the inner **for** loop of line-9 will also run 3 times for j=0 to j=2.

The outer loop therefore scans each row, and for each value of i, the inner loop scans all the columns in that row for the different values of j. Hence the [i, j] combinations produced by the loops for n=3 will be:

- **Loop-1: i=0**: Loop-2: j=0, j=1, j=2. Hence the matrix indices covered will be (0,0), (0,1), (0,2)
- **Loop-1: i=1**: Loop-2: j=0, j=1, j=2. Hence the matrix indices covered will be (1,0), (1,1), (1,2)
- **Loop-1: i=2**: Loop-2: j=0, j=1, j=2. Hence the matrix indices covered will be (2,0), (2,1), (2,2)

When the loops have completed, the input values will get stored in the matrix **mat**. The nested **for** loops from line-14 to 18 are used to access all the elements of the matrix and print them in matrix form. The **printf()** statement in line-16 prints each element accessed. Note how the tab character '\t' is used to print each value at fixed tab distances so that the column values are properly aligned.

The final nested loops from line-19 to 24 are used to get the `sum` of the values for each row. In line-20 after the outer loop starts for a given row the variable `sum` is initialised to 0. The nested inner `for` loop in line-21 then accesses all the values in that row and the code in line-22 adds the row values to the variable `sum`. Once the inner `for` loop ends, the calculated `sum` for that row is printed using the `printf()` statement of line-23.

After the end of each run of the inner loop when the outer loop starts with a new value of i, the `sum` variable is again initialised to 0 in line-20 to remove the sum value stored in it from the previous row.



Passing arrays to functions

- **Passing arrays to functions (without using pointers explicitly):**

Just like normal variables, an entire array can also be passed to a function for calculations with the array elements. However the **method behind the passing of an array is completely different from that of passing any other variable**. As you may recall, while passing a variable to a function, a copy of the actual argument value is made by C and that **copy** is passed to the called function as a formal argument. Therefore any changes made to that value by the called function do not affect the actual variable value.

However when an array is passed to a function, instead of all the elements of the array being passed to the function, **only the address of the first element of the array** is passed. Depending upon the type of the array (i.e. int, float etc.) the subsequent addresses of the elements are determined from the size of the array. Otherwise, it would be really time and space consuming to copy all the elements of an array and then to again pass the copied values back to the called function. Thus if an `int` type array consists of 5 integers and suppose the memory address of the first integer is 6200, then automatically the addresses of the subsequent elements can be calculated as 6202, 6204, 6206, 6208 respectively. Hence it is sufficient to indicate the address of the first element of the array and the total number of elements stored, to access the entire array.

The **address of the first element** can be **passed in two ways**. Either by writing `&arr[0]` where `arr` is the name of the array, or by writing only `arr`. When only the name of an array is used it indicates by default the address of the first element of the array. The formal argument therefore receives the address of this first element of the array (more of this is discussed in the next section).

Moreover since the address of the variables are passed to the called function, **any change made to the array elements happens on the original array and are therefore permanent** i.e. the same changes are reflected anywhere, where the same array is being used. When arguments are passed by this manner then it is known as passing an argument by **reference** as opposed to passing an argument by value. Keeping the above points in mind the **following rules should be followed** while passing an array to a function:



Rules to use an array in a function call

1. **Only the array name should appear without brackets** as an actual argument within the function call. Thus if we have an array called `marks[20]`, then the actual argument when passing this array will be **only** `marks` and **not** `marks[20]` or `marks[]`. Example `display(marks, n);`
2. When declaring **the formal arguments within the function header, the array name is however written with a pair of empty square brackets** (for example as `marks[]`). (For a two-dimensional array however, the column index should be specified, as we will see later).

The program below illustrates the above concept by using a function to calculate the sum of the squares of the numbers in an array.

```
1  /*Program-102: Program to find the sum of the squares of numbers in an array */
2  #include<stdio.h>
3  #define MAX 100
4  int Sum(int list[], int x);       /*Function prototype of Sum function is declared*/
5  void main()
6   { int arr[MAX], i, n, s;
7     printf("\nEnter total number of elements in the array: ");
8     scanf("%d", &n);
9     for(i=0; i<n; i++)
10     {printf("\nEnter Element-%d: ", i);
11      scanf("%d", &arr[i]);
12     }
13    s = Sum(arr, n);
14    printf("\nThe required sum is %d", s);
15  }
```

```
16  int Sum(int list[], int x)
17   {int i, m = 0;
18    for(i=0; i<x; i++)
19       m = m + list[i]*list[i];
20    return m;
21   }
```

Output:

```
Enter total number of elements in the array: 5
Enter Element-0: 3
Enter Element-1: 5
Enter Element-2: 7
Enter Element-3: 9
Enter Element-4: 11
The required sum is 285
```

In line-4, the function prototype to carry out the sum is written. In line-6, an array is declared to store 100 integer values. In line-7, the actual number of elements in the array is entered within the variable `n`. The `for` loop of line-9 is then used to enter the values in the array.

In line-13, the `Sum()` function is called with the arguments `arr`, and `n`. Note that for a linear array, the name of the array is used without the square brackets as the function argument (i.e. neither `arr[]`, nor as `arr[n]` as discussed earlier). Along with the name of the array, the number of elements `n` in the current array is also passed to the function. Without this there are no ways the called function can know about the number of values in the array. Accordingly the header of the function `Sum()` contains the formal argument `list[]` corresponding to the actual argument `arr` (indicating the address of the first element of the array) and the formal argument `x` corresponding to the actual argument `n` (indicating the total number of elements).

In our example, the **address of the value 3** i.e. the address of `arr[0]` will be stored in `list` and the total number of elements i.e. 5 will be stored in the variable `x`. In line-17, the variable `m` is declared to store the sum of the squares. The `for` loop of line-18 is used to calculate the square of the values. Line-20 then finally returns the calculated value `m`. Since `m` is also an integer, the return data type of the function is also an integer as indicated in the function header in line-16.

The returned value is stored within the variable `s` in line-13 and displayed in line-14 using `printf()`.

13.5 Arrays and Pointers



Arrays and Pointers

Till now we have dealt with variable types that deal with integers, floating point numbers, characters etc. The **operating system reserves memory space** in the RAM to store these variables, when variables of these data types are declared. Thus each variable has a particular memory address associated with it. When declaring a variable, a **variable name is given to identify the variable.** That particular variable name is then used to access the value stored in that variable. Thus all calculations or comparisons are made using the name of the variable.

A *pointer* is a variable that stores the address of another variable

However there is another method **to use the value stored in a variable without using its variable name.** The method involves using the memory address of that variable. To access a variable using its memory address, sometimes the **address itself needs to be stored** in another variable.

Due to some special properties of memory addresses, C provides us with a special type of variable called a **'pointer' to store the memory address locations of other variables.** Address value of a variable can be stored in such a pointer type variable only. Pointers are used extensively in C.

There is a **direct relation between a pointer and an array type data.** We will see in this section how we can use an alternative notation using pointers to access an array. In the next class we will have a detailed discussion on using pointers in various other situations.

The Address operator:



The address operator

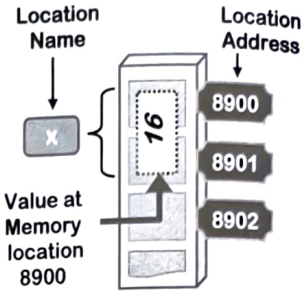
C provides us with **a special operator called the 'address of' operator** represented by the ampersand symbol **&** to access the address of a variable (we have already used this operator while using the `scanf()`

function). The following example shows a method to **view the address location** where the variable **x** is stored (address location indicates the **starting address** of the variable):

The address operator & is used to get the address of a variabe

```
1  /*Program-103:Use of Address Operator*/
2  void main()
3     { int x = 16;
4       printf("\nAddress of x = %u ", &x );      /*Address*/
5       printf("\nThe value of x = %d ", x );     /*Value  */
6     }
```

Address operator used on variable x



Output:

```
Address of x = 8900
The value of x = 16
```

The expression &x returns the address where the value of the variable x is stored. In this case it is found to be the memory location number 8900. Since an address location is always a positive integer, we have used the format specifier %u to display an unsigned integer type variable.

Declaring a Pointer

▪ Declaring a pointer:

Till now we have **only displayed the address location** of a particular variable using the address operator & on the variable name. However, there may be a need to store this address for later use in a program. For this, just like any other variable declaration, one can also **declare a variable to store the address location of another variable**. As discussed in the previous section, such a variable is known as a **pointer variable**.

Data type of a pointer is the data type of the variable whose address it stores.

Being a special type of variable, a **pointer needs to be declared in a special way too**. To declare a pointer variable, we have to first know the type of the variable whose address the pointer will store, i.e. whether the pointer will store the address of an integer type variable, or a float type variable or a character type variable etc. The **data type of that variable whose address it will store**, will form the **data type of the pointer** (the reasons for this will be made clear in later).

Data type of a pointer

Thus we declare a pointer variable by first stating the **data type of the pointer, followed by a ' * '**, followed by the **pointer name**. The following examples will make the idea clear.

- `int *x;` ⇒ declares a pointer variable called **x** which **can store the address of an integer** type variable.
- `float *y;` ⇒ declares a pointer variable called **y** which **can store the address of a float** type variable. It **does not** indicate that **y** is a floating point type variable.
- `double *z;` ⇒ declares a pointer variable called **z** which **can store the address of a double** type variable. It **does not** indicate that **z** is a double type variable.
- `char *m;` ⇒ declares a pointer variable called **m** which **can store the address of a character** type variable. It **does not** indicate that **m** is a character type variable.

In our programs we have **declared pointer variable names starting with a 'p'**, as in **px, py** etc.

Indirection Operator

▪ The Indirection operator:

Till now we have simply used the variable name to display or access the value stored in a particular memory location. Now we will find out how to access the same value using its memory address location, instead of its name. We have to use a special operator called the **indirection operator (*) to access the value of a variable using its address**. The indirection operator **acts on an address** and displays the value that is stored in that particular address location.

The Indirection Operator gets the value at the address stored in a pointer.

In the previous example where the variable **x** has address location '8900', the indirection operator can work on this address and access the value that is stored there i.e. 16. The following example shows the **use of the indirection operator** to access the value contained in the variable **x**.

```
/*Program-104: Use of the indirection operator*/
void main()
{ int x = 16;
  printf("\nAddress of x = %u ", &x );
  printf("\nThe value of x using variable name = %d ", x );
  printf("\nThe value of x using indirection operator = %d ",*(&x) ); /*Value at addr*/
}
```

Indirection operator used on address of the variable x

Output of the above program is:

```
Address of x = 8900
The value of x using variable name = 16
The value of x using indirection operator = 16
```

Since `&x` represents the address where value of the variable `x` is stored, the `*` or **indirection operator** when placed before the address i.e. before `&x` will **give the value stored at the address location** `&x`. In the above case, the value stored at the address location of `x` is `16`, as shown.

Thus the **indirection operator** can also be called the **value-pointed-by-address** operator or **value-stored-at-address** operator as it can be used to display the value stored at a particular address location using that address location. The following diagram clarifies the idea.

Value pointed by Address

Thus:

```
                x  ⇒  16
               &x  ⇒  8900
* (&x) ⇒ *(8900)  ⇒  16
```

⇐ Value of variable x
⇐ Address of variable x
⇐ Value stored at address of variable x

Value stored at | Address location of x | Value stored at address location 8900

When we had run the above program, we got the address location for the variable `x` as `8900`. However **this is not a fixed value** and will change from computer to computer and within the same computer for each run of the program, because each time a variable is declared, it is assigned a block of address depending upon the available memory slots at that time. Thus we can conclude that the **address of a variable is itself a variable and hence can be stored in another variable.**

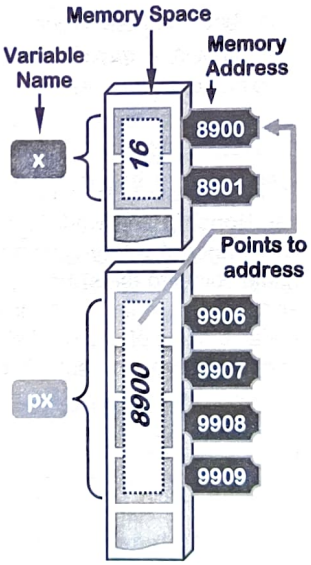
Using a pointer variable along with the indirection operator:

Pointer and Indirection operator

In this section let us see how we can use a pointer variable to store the address of another variable and then access the value of that variable using the pointer and the indirection operator. The program in the next page illustrates the **use of pointers** to address and display variables:

```
/*Program-105: Declaring and using a Pointer*/
void main()
  {int x, *px;
   px = &x;
   x = 16;
   printf("\nAddress of x using address operator = %u", &x );
   printf("\nAddress of x using pointer = %u", px );
   printf("\nThe value of x using variable name = %d", x );
   printf("\nThe value of x using address = %d", *(&x) );
   printf("\nThe value of x using pointer = %d", *px );
  }
```

The * indicates that px is not a normal integer variable but a pointer type variable



The declaration `int *ptr;` declares a pointer ptr that can point to the address of an integer type data.

Output of the above program is:

```
Address of x using address operator = 8900
Address of x using pointer = 8900
The value of x using variable name = 16
The value of x using address = 16
The value of x using pointer = 16
```

The diagram in the previous page shows the relation between the different variables, and the values stored in those variables. It is seen that the memory location indicated by the variable name **x** has the address **8900**. The pointer variable **px** is stored at the memory location **9906**. (We have reserved **4 bytes** for pointer variables, as is done in modern ANSI compilers).

Meaning of int *px

Line number 3 in the above program is used to declare the variables **x** and **px**. The variable **x** is a normal integer variable while the variable **px** is a pointer variable. To indicate that **px** is a pointer variable, as stated in the previous page, a '*' is placed before **px**. Therefore the declaration **int *px indicates that px is a pointer type variable that can store the address of an integer type variable**.

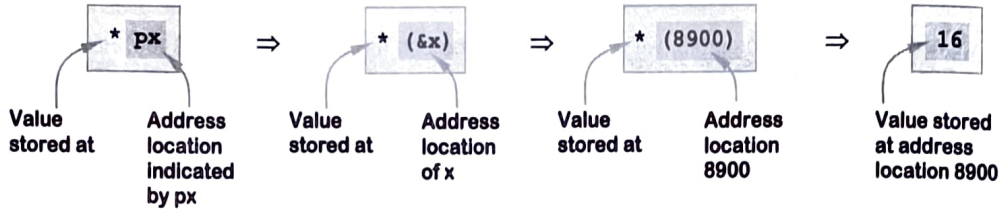
In line number 4 the address of the integer variable **x** is assigned to the pointer **px**. The address operator is used to get the address of the integer variable **x**. Thus **&x** gives the address location where the variable **x** is stored. In this case it is equal to **8900**. The statement **px=&x** then basically **assigns the address location of x, i.e. 8900 to the pointer variable px**.

Since the value **16** is assigned to the variable **x** in the program in line 5, this value gets stored in the memory location **8900**. Similarly, the address of the variable **x** is assigned to the pointer variable **px** in line 4, and **hence this address i.e. the value 8900 gets stored in the memory location 9906**.

Lines 6 and 7 both are used to display the address of the variable **x**. While in line 6 the address operator '&' is used to extract the address of **x** and display it, in line 7 the pointer variable **px** is used to display the address. Since **px** stores the value **8900** it gets displayed by the **printf()**.

Lines 8, 9, and 10 are used to display the content of the variable **x** i.e. **16**. In line 8 the value is displayed by simply using the variable name **x**. In line 9, first the address of **x** is extracted by using the '&' operator. Next the value stored at that address i.e. at **8900** is displayed by using the indirection operator '*'. **In line 10, the indirection operator '*' acts on the pointer px** to display the same value stored at the address pointed by **px** i.e. it displays the value stored at **8900**.

Meaning of *px in line 10



There can be a confusion regarding the use of the indirection operator '*' in line 3 and line 10. Note that during the declaration in line 3, the indirection operator '*' in the statement **int *px** can be interpreted to indicate that "**the value stored at address in px is of type int**". Whereas in line 10, the '*' before **px** is simply used to indicate the **value stored at address in px**.

The above process of getting the value of the variable **x** by using the indirection operator '*' on the pointer **px** is known as **dereferencing a pointer**. We always need to dereference a pointer to get to the value that it points to, as has been done in line 10 of the previous program.

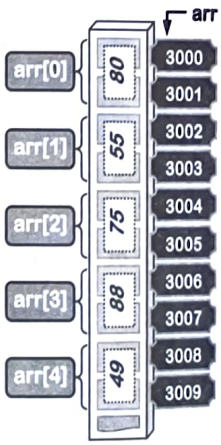
Relation between pointer and array

- **Relation between a Pointer and an Array:**

It is interesting to know that in C, **arrays and pointers share a special relationship** and work in a similar manner. This is because the **name of a one-dimensional array gets automatically converted to a pointer** and the array name points to the **starting element** of the array. This may make you think that an array and a pointer are identical – but in reality these are **not** and a proper understanding of this can help you avoid various confusions in future. Let us consider an integer array to clarify this and see how its behaviour is related to a pointer.

```
int arr[5] = {80, 55, 75, 88, 49};
```

Once the above array is declared and initialised, memory space is allocated for storing the 5 integer values as usual. The memory diagram shows the memory allocation. The array starts from the memory location **3000** and goes up to **3009** i.e. a total of **10 bytes**. The contents of the locations are also displayed.



Under such a situation, as per the C standard, the name of the array i.e. `arr`, **without any square brackets** refers to the address of the 0th element of the array i.e. to the **address of** `arr[0]` i.e. `&arr[0]`. Hence:

Array name by itself i.e. arr is same as &arr[0]

Therefore in this example the array name `arr` is referring to the address location `3000`. We have seen that a variable which points to an address is a pointer. **Therefore the name of an array can be treated as a pointer**. However it should be noted that an **array name is not same as a pointer**, though they may behave in a similar manner. Thus, though `arr` points to address `3000`, but since the location of the array (i.e. the block of memory allocated to the array) is fixed during run time, **this address is a constant and cannot be changed**. On the other hand, a pointer can be used to store any address value during run time, as long as it is of same data type (refer to program-105). Hence for the declaration `int *parr, arr[5];` we can write `parr=arr;` to store the address of the starting element of the array within the pointer `parr`, but we can never write `arr=parr;` as the address of the first element of the array implied by `arr` cannot be assigned a different address once the array `arr[]` has been declared. Thus we find that **pointers and arrays are not exactly the same**. Don't feel confused, read the section again to understand the concept!

Meaning of array name

The name of an array indicates the address of the first element of the array

Let us now discuss the various arithmetic operations on pointers and what they mean. We have seen that when an array is declared, the array name points to the location of `arr[0]`. For any array `arr[]` we therefore have:

```
arr     ⇒   &arr[0]
```

The notation arr ≡ &arr[0] for an array arr

However, when we **add '1' to the array name** `arr`, **it indicates the address of the next element i.e. element** `arr[1]` (and not the address of the next byte). Similarly when we add '2' to the array name `arr`, it indicates the address of the element `arr[2]`. If we generalise this property, we have:

```
arr       ⇒   &arr[0]   ⇒   location 3000   (As per the C standard)
arr + 0   ⇒   &arr[0]   ⇒   location 3000   (As arr and arr+0 indicate the same thing)
arr + 1   ⇒   &arr[1]   ⇒   location 3002   (NOT 3001)
arr + 2   ⇒   &arr[2]   ⇒   location 3004   (NOT 3002)
∴ arr + i ⇒   &arr[i]   ⇒   Address of the ith element of the array
```

Hence in general (arr+i) ⇒ &arr[i] ⇒ Address of the ith element of the array

Meaning of (arr+i)

We have seen that the array name `arr` indicates the address location `3000`. When we add '1' to `arr` the result is not `3001` but `3002`. That is, the value '2' gets added to `3000`. This is a speciality of arithmetic operations on address locations and is called **pointer arithmetic**. Pointers and arrays behave in a similar manner as far as pointer arithmetic is concerned. Thus we have the property:

Pointers increment in units depending on the data type they point to

The notation arr+i ≡ &arr[i] for an array arr

Since an `int` type data occupies two bytes, **with each increment an** `int` **pointer will increment by units of two bytes** to point to the next data element in a list. Whereas a `float` type pointer will increment by units of four bytes to point to the next data element. Similarly when dealing with arrays, when using a `float` type data array `arr[]`, if `arr` points to `3200`, then `arr+1` will indicate `3204` and **not** `3201`. That is, `arr+1` points to the address of `arr[1]`. Hence for an array, **when an integer i is added to the array name, the resultant expression points to the ith array element**.

One can store a set of values of a particular data type in **two different ways**. The first method is by using an array (static array), which needs to be declared during writing the program code. The other is to reserve a memory block sufficient enough to hold all the values, but this time while running the program (dynamic array) (*will be discussed in class 12*). If the end result is the same in both the methods, i.e. storing a set of values of a similar data type, then these should also behave in a similar manner. As the array name with proper subscript is used to access the stored values in the first method (as `arr[0]`, `arr[1]` etc.), in the second method a pointer is used to first point to the address of the first value stored in the memory block. Then by successively incrementing the pointer using an expression like `p+1`, where `p` is the pointer, one can access all the values stored in the memory block one by one. However, in this method if the pointer was not declared with a data type (as `int *p` etc.), the meaning of the statement `p+1` would be vague. It will not be clear how many bytes to jump for such an increment to point to the next value in the block. Hence pointers should **be declared based on the data type whose address it is going to point**.

The notation *(arr+i) ≡ arr[i] for an array arr

We have seen that the array name `arr` points to the address of the starting element of the array `arr[]`. Thus `*arr` **indicates the value stored at the 0th position of the array**. However, as per array notation `arr[0]` represents the value at the 0th position of the array. Therefore `*arr` ⇒ `arr[0]`. Now `*arr` is same as `*(arr+0)`. In a similar manner we can represent the 4th element of the array pointed by the address `(arr+4)` by the expression `*(arr+4)` ⇒ `arr[4]`. In general:

`*(arr+0)` ⇒ `*(&arr[0])` ⇒ Value at the address `&arr[0]` ⇒ `arr[0]`
`*(arr+4)` ⇒ `*(&arr[4])` ⇒ Value at the address `&arr[4]` ⇒ `arr[4]`
`*(arr+i)` ⇒ `*(&arr[i])` ⇒ Value at the address `&arr[i]` ⇒ `arr[i]`

Hence *(arr + i) is same as arr[i]

Meaning of *(arr+i)

The fact is that, whenever we write a statement like `arr[i]`, the C compiler immediately converts it using pointer notation to the form `*(arr+i)`. Hence the following statements are all same, as after they are converted to the above notation by the compiler, they all **represent the same thing**:

`arr[i]` same as `*(arr + i)` same as `*(i + arr)` same as `i[arr]`

▪ Arithmetic and Logical Operations on a Pointer:

Operations on Pointers

We can perform the following operations on pointers. Thus for a pointer `p` of data type `dt` we have:

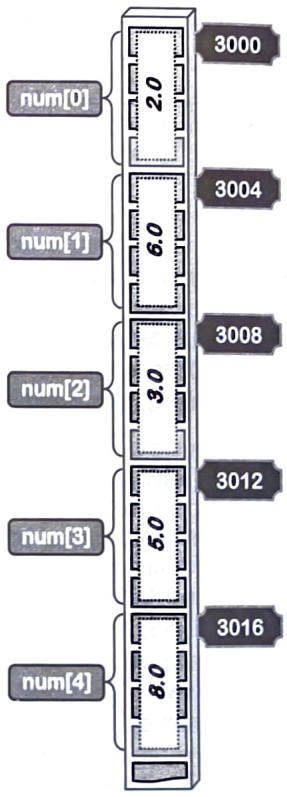
- **Add an integer to a pointer** as `(p+i)`: Points to address `p+i*sizeof(dt)` ahead of `p`
- **Subtract an integer from a pointer** as `(p-i)`: Points to address `p-i*sizeof(dt)` behind `p`
- **Subtract one pointer from another** `(p1-p2)`: Gives number of bytes between `p2` and `p1`
- **Compare two pointer variables**: To find which address is ahead of the other.

You can add an integer to a pointer or subtract two pointers.

However following **operations are not possible on pointers** for obvious reasons:

- Add two pointers
- Multiply two pointers
- Multiply a pointer with a number
- Divide two pointers
- Divide a pointer with a number

We had seen in the previous section that **adding an integer to a pointer moves the pointer to a memory location ahead of the pointer** depending upon the data type it points to. Similarly, subtracting an integer from a pointer will make the pointer point to a previous memory location.



Why subtract two pointers

However **to subtract** one pointer from another, it needs to be assured that **both the pointers point to elements of the same array** or memory block. The difference indicates the number of elements separating the two array elements to which the two pointers point.

▪ Manipulating Arrays using Pointer Notations:

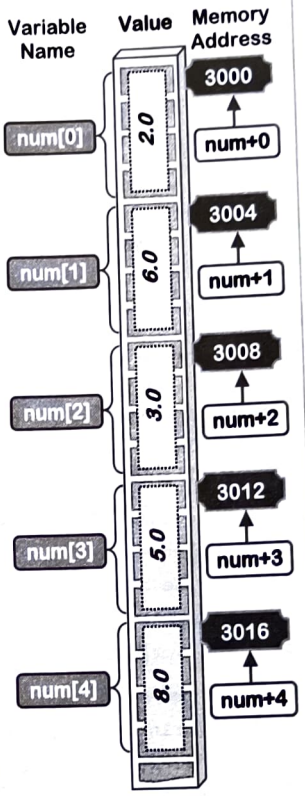
The following program illustrates the use of **pointer notation to access array elements** by calculating the standard deviation of a set of numbers.

```
1  /*Program-106: Standard Deviation using Pointer Notation*/
2  #include<stdio.h>
3  #include<math.h>
4  #define MAX 5
5  void main()
6    {float num[MAX], sum1=0.0, sum2=0.0, term, mean, sd;
7     int i;
8     for(i=0; i<MAX; i++)
9       {printf("\nEnter number-%d: ", i+1);
10       scanf("%f", num+i);      /*Enter data at address num+i  */
11       sum1 = sum1 + *(num+i); /*Value stored at address num+i*/
12      }
```

```
13      mean = sum1/MAX;
14      for(i=0; i<MAX; i++)
15       (term = pow( (*(num+i)-mean),2 ); /*Access data at num+i*/
16        sum2 = sum2 + term;
17       )
18      sd = sqrt(sum2/MAX);
19      printf("\nThe required standard deviation is %.3f", sd);
20     )
```

Output of the above program:

```
Enter number-1: 2
Enter number-2: 6
Enter number-3: 3
Enter number-4: 5
Enter number-5: 8
The required standard deviation is 2.135
```



In line-4 we have defined the value `MAX` as 5. In line-6, the floating point array `num[MAX]` have been declared along with other variables used in the program.

In line-8, a `for` loop is used to enter the values into the array. The loop runs from `i=0` to `i<MAX`. Inside the loop in line-10, the pointer notation is used instead of the array notation to enter a value into the ith position of the array, using the `scanf()` statement. Remember that the array name `num` indicates the address of the 0th element of the array i.e. `&num[0]` (or 3000 in this case). Therefore `num+i` indicates the address of the ith element of the array i.e. `&num[i]`. Hence we have replaced the notation `&num[i]` with `num+i` in `scanf()`. The memory diagram shows the position of `num+i` for different values of `i`.

In line-11, the sum of the entered numbers is calculated again by using the pointer notation. As `(num+i)` indicates the address of the ith element of the array, `*(num+i)` indicates the value stored at the address location indicated by `(num+i)` or simply the value `num[i]`. In line-13 the `mean` of the above values is calculated.

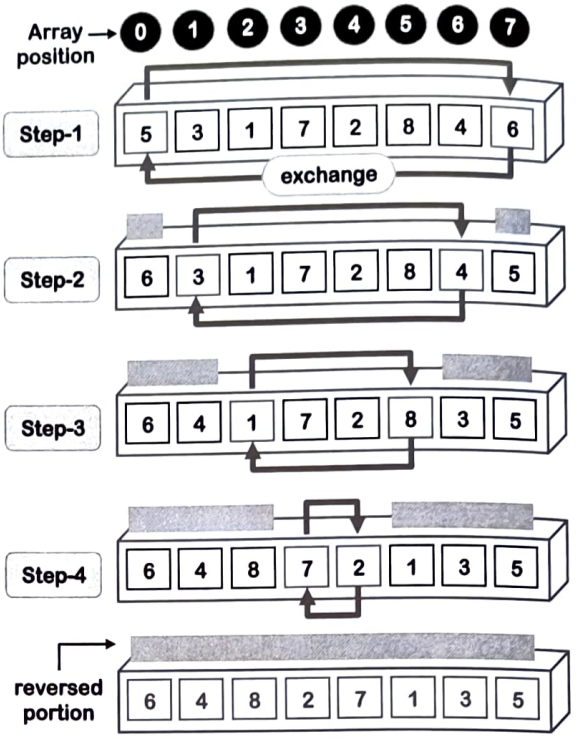
Line-14 starts another `for` loop, to calculate the sum of the square terms. In line-15, the square of the expression `(*(num+i)-mean)` or `(num[i]-mean)` is calculated using the `pow()` function. In line-16 the sum of these square terms is calculated. Finally line-18 calculates the required standard deviation using the `sqrt()` function.

13.6 Some worked out examples

Various problems can be handled easily by using arrays as listed below:

- Array order **reversal** i.e. reversing the contents of an array
- Finding the **maximum** and **minimum** numbers in a set of numbers
- **Inserting** and **removing** specific elements from an array
- **Sorting** of numbers i.e. arranging number in ascending or descending order. It can be done using:
  - **Bubble**-sort method
  - **Selection**-sort method
  - Insertion-sort, Shell-sort, Quick-sort methods etc. [Not in syllabus]
- **Searching** of numbers from the array. It can be done using:
  - **Linear** search method
  - **Binary** search method
- Generation of **histogram** distribution

We will now discuss some of the above problems and write down the necessary code and explain them.



In this section let us consider the simple problem of **reversing the contents of an array**, i.e. if we have an array with **n** elements then our task will be to do the following:

Assign the value of `A[0]` to `A[n-1]`, `A[1]` to `A[n-2]`, ... `A[n-2]` to `A[1]`, and `A[n-1]` to `A[0]`. This involves basically exchanging the values stored in these pair of elements. That is, exchange `A[0]` with `A[n-1]`, `A[1]` with `A[n-2]`, and so on.

Let us analyse the problem with an **8 element integer array** as shown to the right. To achieve the reversal, we will have to first exchange the contents of the positions `A[0]` and `A[7]`. Then we exchange the values in the positions `A[1]` and `A[6]` and so on.

This process continues till we reach the middle of the array by which time all elements would have been exchange. The diagram on the right clarifies the point. Hence the number of steps to carry out the exchange will be **s = 8/2**, where **8** signifies the number of elements in the array. For odd number of elements say n=7, s = 7/2 = 3 (taking **integer division**) i.e. the exchange will take place up to 3 steps. This will not pose any problem as the centre element i.e. the 4th element in that case does not require any shuffling.



Reversing an array

```
1  /*Program-107: To reverse the contents of an array*/
2  #include<stdio.h>
3  #define MAX 8
4  int main()
5  {int num[MAX], i, j, temp;
6   for(i=0; i<MAX; i++)
7      {printf("Enter Element-%d: ", i);
8       scanf("%d", &num[i]);
9      }
10  for( i=0, j=(MAX-1); i<(MAX/2); i++, j-- )
11     {temp = num[i];
12      num[i] = num[j];
13      num[j] = temp;
14     }
15  for(i=0; i<MAX; i++)
16     printf("\nElement-%d of reversed array: %d", i, num[i]);
17  return 0;
18 }
```

Output:

```
Enter Element-0: 2
Enter Element-1: 5
... ... ...
Enter Element-6: 15
Enter Element-7: 17
Element-0 of reversed array: 17
Element-1 of reversed array: 15
... ... ...
Element-6 of reversed array: 5
Element-7 of reversed array: 2
```

In this program the values are entered into the array `num[]` using the `for` loop of line-6. The `for` loop of line-10 is then used to reverse the contents of the array. Note that the index of the counter `i` and `j` are initialised using comma operator. `i` is initialised to 0 and `j` is initialised to `(MAX-1)` = (8-1) = 7, so that the initial exchange takes place between the values `num[0]` and `num[7]`. The condition inside the `for` loop is also `i<(MAX/2)` and **not** `i<=(MAX/2)` so that the last exchange takes place between `num[3]` and `num[4]`.

The next program finds the **maximum value from a list of numbers**, without sorting the array.



To find the maximum from an array

```
1  /*Program-107: To find the maximum from a list of numbers in an array*/
2  #include<stdio.h>
3  #define MAX 8
4  void main()
5  {int num[MAX], i, maximum;
6   for(i=0; i<MAX; i++)
7     {printf("\nEnter Element-%d: ", i);
8      scanf("%d", &num[i]);
9     }
10  maximum = num[0];
11  for( i=1; i<MAX; i++ )
12    { if( num[i] > maximum )
13          maximum = num[i];
14    }
15  printf("\nThe maximum value is %d", maximum);
16 }
```

Output:

```
Enter Element-0: 5
Enter Element-1: 3
Enter Element-2: 12
_ _ _
Enter Element-7: 8
The maximum value is 12
```

An integer array is defined in line-5 to store 8 values. These values are then entered using the `for` loop of line-6. Next the value stored in the position `num[0]` is copied to the variable `maximum` in line-10. The `for` loop of line-11 is then used to compare this value stored in the variable `maximum` with the rest of the values in the array one by one starting with `num[1]`. The `if` statement of line-12 does this comparison. After the comparison if it is found that the number stored at `num[i]` for a particular position `i`, is larger than the value stored in `maximum` then in line-13 the value stored in `maximum` is replaced by this value. The final `maximum` value is printed in line-16.

The next program is used to **insert a value at a particular position within an existing array**.



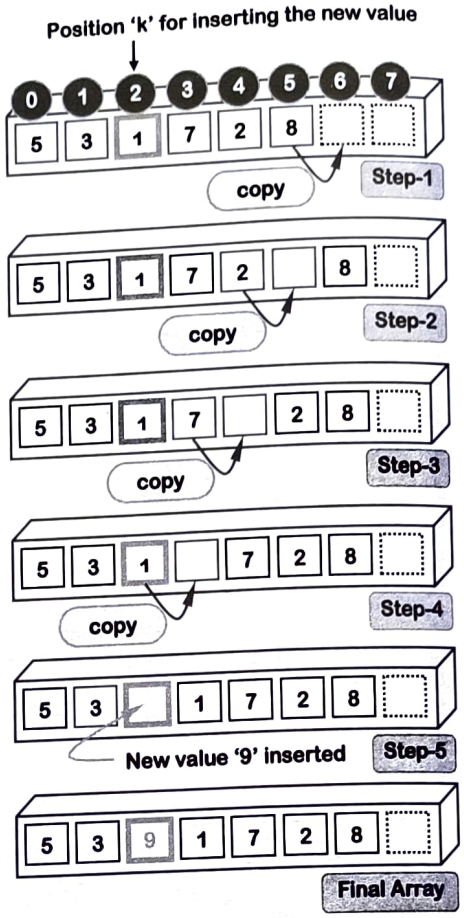
Inserting a value into an existing array

```
1  /*Program-108: Inserting a value at k-th position*/
2  #include<stdio.h>
3  #define MAX 100
4  void main()
5  {int arr[MAX], n, i, num, k;
6   printf("\nEnter total number of elements: ");
7   scanf("%d", &n);
8   for(i=0; i<n; i++)
9     {printf("\nEnter Element-%d: ", i);
10     scanf("%d", &arr[i]);
11    }
12  printf("\nEnter new value to insert: ");
13  scanf("%d", &num);
14  printf("\nEnter position at which to insert: ");
15  scanf("%d", &k);
```

```
16   for(i=(n-1); i>=k; i--)
17     arr[i+1] = arr[i];
18   arr[k] = num;
19   for(i=0; i<(n+1); i++)
20     printf("\nAfter insertion %d", arr[i]);
21   }
```

Output:

```
Enter total number of elements: 6
Enter Element-0: 5
Enter Element-1: 3
… … …
Enter Element-5: 8
Enter new value to insert: 9
Enter position at which to insert: 2
After insertion 5
After insertion 3
After insertion 9
… … …
After insertion 8
```



An integer array to store 100 elements is declared in line-5. In line-7 the user enters in the variable **n** the actual number of elements that he wants to store. These **n** values are entered using the **for** loop of line-8. The new value to insert into this existing array is then entered in line-13 within the variable **num**. Next, in line-15 the position at which to insert this new value is entered into the variable **k**. In the above example we have first entered *6* values into the array i.e. {5, 3, 1, 7, 2, 8} and a new value *9* is inserted into the position with index *2*.

To insert this new value, first space needs to be created at position 2 so that the original value (1, in this example) stored at that position is not lost. To do this, the values are shifted from the end as shown in the diagram above, using the **for** loop of line-16. So the value stored at position **arr[i]** is copied to **arr[i+1]**. This shifting continues till **i=k**, when in this case the value '1' is shifted from position **k** i.e. 2 to position 3. Note how the value of **i** is decreased from **(n-1)** to **k** in the **for** loop. Finally the new value stored in **num** is inserted at position **k** in line-18 outside the **for** loop.

The next program is used to **delete a value at a particular position within an existing array**.



Deleting a value at the k[th] position of an array

```
1   /*Program-109: Deleting a value at k-th position*/
2   #include<stdio.h>
3   #define MAX 100
4   void main()
5   {int arr[MAX], n, i, k;
6    printf("\nEnter total number of elements: ");
7    scanf("%d", &n);
8    for(i=0; i<n; i++)
9      {printf("\nEnter Element-%d: ", i);
10      scanf("%d", &arr[i]);
11     }
12   printf("\nEnter position at which to delete value: ");
13   scanf("%d", &k);
14   for( i=k; i<n-1; i++ )
15      {arr[i] = arr[i+1];}
16   printf("\nArray after deletion of element:");
17   for(i=0; i<(n-1); i++)
18     printf("\n%d", arr[i]);
19  }
```

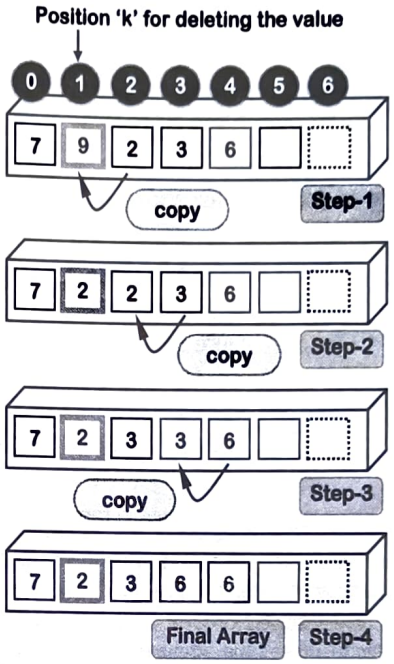
Output:

```
Enter total number of elements: 5
Enter Element-0: 7
Enter Element-1: 9
Enter Element-2: 2
Enter Element-3: 3
Enter Element-4: 6
Enter position at which to delete value: 1
Array after deletion of element:
7
2
3
6
```

An integer array to store 100 elements is declared in line-5. In line-7 the user enters in the variable `n` the actual number of elements that are to be stored. These `n` values are entered using the `for` loop of line-8. The position at which to delete the value is entered in line-13. In the above output example we have entered *5* values into the array i.e. {7, 9, 2, 3, 6} and the value at index *1* is to be deleted.

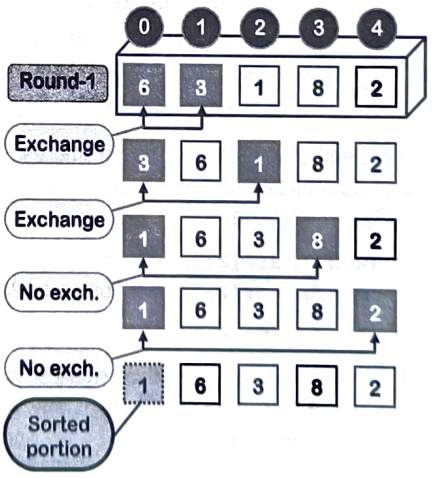
To delete the value at position [1] we will basically overwrite the value at position [1] with the value at position [2]. Next we will overwrite the value at position [2] with the value at position [3] and so on, until the last value in the array at position [n-1] is copied to the second last position in the array i.e. [n-2].

In our example, we have to delete the value at position `k=[1]`. Hence the loop starts with `i=k` and the value at position [i+1] is copied to the position [i], i.e. the value at position [2] is copied to position [1] (refer to diagram on the right). The process continues till `i=n-2`, when the last value at position [n-1] is copied to position [n-2]. In our example the value at position [4] is copied to position [3]. Note that the last value (6 in this case) though shifted to the second last position, still remains in the last position as shown in the diagram. Hence the array is accessed in line-17 up to position [n-2] (i.e. `i<n-1`)



The next program is used to **sort a list of numbers** in ascending order using **Selection Sort method.**

```
1  /*Program-110: Selection Sort Method*/
2  #include<stdio.h>
3  #define MAX 100
4  void main()
5  {int arr[MAX], i, j, n, temp;
6   printf("\nEnter total number of elements: ");
7   scanf("%d", &n);
8   for(i=0; i<n; i++)
9     {printf("\nEnter Element-%d: ", i);
10     scanf("%d", &arr[i]);
11    }
12  for(i=0; i<n-1; i++)
13    for(j=i+1; j<n; j++)
14      {if(arr[i]>arr[j])
15         {temp = arr[i];
16          arr[i] = arr[j];
17          arr[j] = temp;
18         }
19      }
```

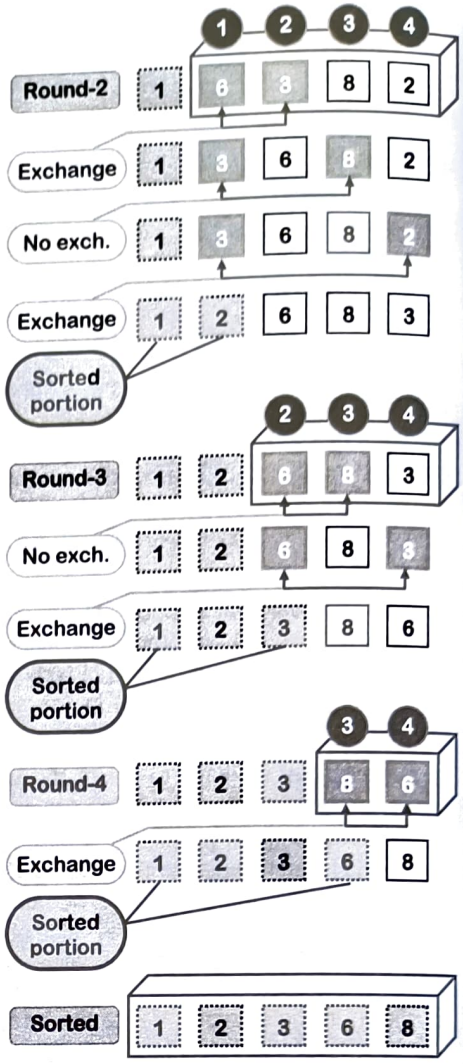



Selection Sort Method

```
20   for(i=0; i<n; i++)
21      printf("\nSorted array[%d] = %d", i, arr[i]);
22   }
```

Output:

```
Enter total number of elements: 5
Enter Element-0: 6
Enter Element-1: 3
Enter Element-2: 1
Enter Element-3: 8
Enter Element-4: 2
Sorted array[0] = 1
Sorted array[1] = 2
Sorted array[2] = 3
Sorted array[3] = 6
Sorted array[4] = 8
```



The technique of selection sort is explained with the help of the above example and the diagram on the right. The array is used to store 5 values (**n=5**). A list of values is entered using the `for` loop of line-8. Let the list of values be {6,3,1,8,2} which is to be **sorted in ascending order**.

Round-1:a) `arr[0]` i.e. 6 is compared with `arr[1]` i.e. 3. Since 6>3 (**niao**[↑]), the values are **exchanged**

b) `arr[0]` i.e. 3 is compared with `arr[2]` i.e. 1. Since 3>1 (**niao**), the values are **exchanged**

c) `arr[0]` i.e. 1 is compared with `arr[3]` i.e. 8. Since 1<8 (**iao**[↑]), **no exchange** takes place

d) `arr[0]` i.e. 1 is compared with `arr[4]` i.e. 2. Since 1<2 (**iao**), **no exchange** takes place

The values in the list are now: **1, 6, 3, 8, 2**

Thus at the end of round1, the smallest value in the list i.e. **1** is occupying position `[0]` in the list, i.e. it has been put in the correct sorted order. To sort the remaining part the array the same process is again repeated with the rest of the array i.e. from element index [1] to [4] (see diagram on the right).

Round-2:a) `arr[1]` i.e. 6 is compared with `arr[2]` i.e. 3. Since 6>3 (**niao**), the values are **exchanged**

b) `arr[1]` i.e. 3 is compared with `arr[3]` i.e. 8. Since 3<8 (**iao**), **no exchange** takes place

c) `arr[1]` i.e. 3 is compared with `arr[4]` i.e. 2. Since 3>2 (**niao**), the values are **exchanged**

The values in the list are now: **1, 2, 6, 8, 3**

At the end of round2, the second smallest value in the list i.e. 2 is occupying position `[1]` in the list i.e. it has been put in its correct sorted order. The same process is again repeated now starting from element index [2] to [4] (see diagram).

Round-3:a) `arr[2]` i.e. 6 is compared with `arr[3]` i.e. 8. Since 6<8 (**iao**), **no exchange** takes place

b) `arr[2]` i.e. 6 is compared with `arr[4]` i.e. 3. Since 6>3 (**niao**), the values are **exchanged**

The values in the list are now: **1, 2, 3, 8, 6**

At the end of round3, the third smallest value in the list i.e. 3 is occupying position `[2]` in the list i.e. it has been put in its correct sorted order. The same process is again repeated now starting from element index [3] to [4] (see diagram).

Round-4:a) `arr[3]` i.e. 8 is compared with `arr[4]` i.e. 6. Since 8>6 (**niao**), the values are **exchanged**

The values in the list are now: **1, 2, 3, 6, 8**

We find that at the end of round4, all the elements have been put in their correct sorted order and hence the array gets **finally sorted in ascending order**.

[↑] (**niao**) = **n**ot **i**n **a**scending **o**rder; (**iao**) = **i**n **a**scending **o**rder

Note that for an array with 5 elements we required a maximum of 4 rounds to sort the array. In general **you require a maximum of (n-1) rounds to sort an array with n number of elements.**

In program-110, in line-7, the user enters the number of elements in the array within the variable `n`. In this example `n=5`. The outer `for` loop starting from line-12 is used to count the number of rounds required to sort the array. We have seen that for an '`n`' element array we require a maximum of `(n-1)` rounds to sort the array. Hence the loop runs from `i=0` to `i<(n-1)` i.e. a total of `(n-1)` rounds.

The inner `for` loop starts from line-13. This loop is used for comparing the rest of the elements of the array with a particular element `arr[i]` of the array. We have seen that in round-1, the value 6 in position `arr[0]` was compared with the values starting from `arr[1]`. Next, in round-2, the value stored in position `arr[1]` was compared with the values starting from `arr[2]`. Thus in general, the value in position `arr[i]` is compared with values starting from `arr[i+1]`. Hence range of counter variable `j` is from `j=(i+1)` to `j<n`.

The `if` statement of line-14, is used to compare the element `arr[i]` with `arr[j]` for each value of `j`. In the above program we are arranging the values in ascending order. Therefore in case `arr[i]` and `arr[j]` are in ascending order, then no exchange takes place. However if the values are not in ascending order, then the values are exchanged to put them in proper order. In this way, a particular `arr[i]` is taken and compared with all values after it, by changing the value of `j`.

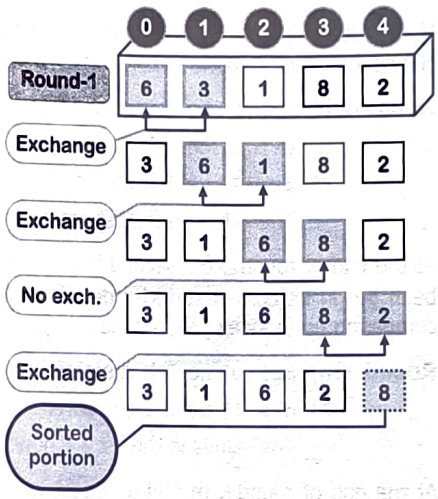
Finally the sorted array is printed using the `for` loop of line-20.

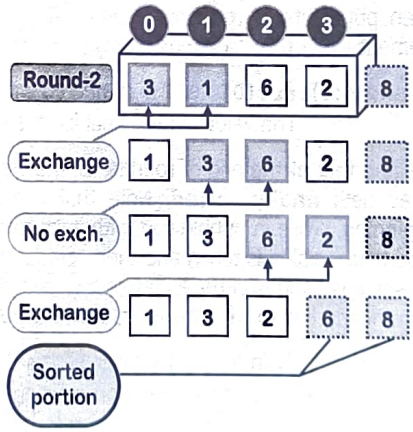
The next program is used to sort a list of numbers using **another sorting method** called **Bubble Sort.**



Bubble Sort Method

```
1  /*Program-111: Bubble Sort Method*/
2  #include<stdio.h>
3  #define MAX 100
4  void main()
5  {int arr[MAX], i, j, n, temp;
6   printf("\nEnter total number of elements: ");
7   scanf("%d", &n);
8   for(i=0; i<n; i++)
9     {printf("\nEnter Element-%d: ", i);
10     scanf("%d", &arr[i]);
11    }
12  for(i=0; i<n-1; i++)
13    for(j=0; j<(n-i-1); j++)
14      {if(arr[j]>arr[j+1])
15         {temp = arr[j];
16          arr[j] = arr[j+1];
17          arr[j+1] = temp;
18         }
19      }
20  for(i=0; i<n; i++)
21    printf("\nSorted array[%d] = %d", i, arr[i]);
22 }
```

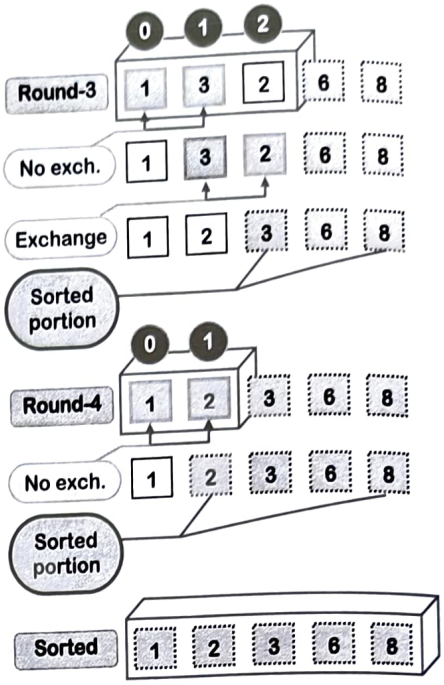




Output:

```
Enter total number of elements: 5
Enter Element-0: 6
Enter Element-1: 3
Enter Element-2: 1
Enter Element-3: 8
Enter Element-4: 2
Sorted array[0] = 1
Sorted array[1] = 2
Sorted array[2] = 3
Sorted array[3] = 6
Sorted array[4] = 8
```

The technique of bubble sort is explained with the help of the above example and the diagram on the right. The array is used to store 5 values (**n=5**). A list of values are entered using the `for` loop of line 7. Let the list of values be {6, 3, 1, 8, 2} which are **sorted in ascending order**. Unlike selection sort, in this technique always **two consecutive elements** are compared.



Round-1:a) `arr[0]` i.e. 6 is compared with `arr[1]` i.e. 3. Since 6>3 (**niao***), the values are **exchanged**

b) `arr[1]` i.e. 6 is compared with `arr[2]` i.e. 1. Since 6>1 (**niao**), the values are **exchanged**

c) `arr[2]` i.e. 6 is compared with `arr[3]` i.e. 8. Since 6<8 (**iao***), **no exchange** takes place

d) `arr[3]` i.e. 8 is compared with `arr[4]` i.e. 2. Since 8>2 (**niao***), the values are **exchanged**

The values in the list are now: **3, 1, 6, 2, 8**

Thus at the end of round1, the largest value in the list i.e. **8** is occupying position `[4]` in the list, i.e. it has been put in the correct sorted order. The same process is again repeated, but this time with the rest of the array i.e. starting from element index [0] up to [3] (see diagram above).

Note that the largest element in bubble sort occupies the last position after a particular round of comparisons. Hence for the next round of comparisons, **we are not considering the already sorted last element.**

Round-2:a) `arr[0]` i.e. 3 is compared with `arr[1]` i.e. 1. Since 3>1 (**niao***), the values are **exchanged**

b) `arr[1]` i.e. 3 is compared with `arr[2]` i.e. 6. Since 3<6 (**iao***), **no exchange** takes place

c) `arr[2]` i.e. 6 is compared with `arr[3]` i.e. 2. Since 6>2 (**niao***), the values are **exchanged**

The values in the list are now: **1, 3, 2, 6, 8**

At the end of round2, the second largest value in the list i.e. 6 is occupying position `[3]` in the list i.e. it has been put in its correct sorted order. The same process is again repeated, now starting from index [0] and comparing up to index [2] (see diagram).

Round-3:a) `arr[0]` i.e. 1 is compared with `arr[1]` i.e. 3. Since 1<3 (**iao***), **no exchange** takes place

b) `arr[1]` i.e. 3 is compared with `arr[2]` i.e. 2. Since 3>2 (**niao**), the values are **exchanged**

The values in the list are now: **1, 2, 3, 6, 8**

At the end of round3, the third largest value in the list i.e. 3 is occupying position `[2]` in the list i.e. it has been put in its correct sorted order. The same process is repeated for the final time now, starting from element index 0 to 1 (see diagram).

Round-4:a) `arr[0]` i.e. 1 is compared with `arr[1]` i.e. 2. Since 1<2 (**iao***), **no exchange** takes place

The values in the final list are now: **1, 2, 3, 6, 8**

We find that at the end of round4, all the elements have been put in their correct sorted order and hence the array gets **finally sorted**. Note that in selection sort with each round the array gets sorted from the beginning, while in bubble sort with each round the array gets sorted from the end. Moreover in selection sort a particular value is taken and then compared with the rest of the values following it. While in bubble sort two consecutive values in the list are compared. We find that for an array with 5 elements a maximum of 4 rounds were needed to sort the array i.e. **a maximum of (n-1) rounds to sort an array with n elements.**

In program-111, in line-7, the user enters the number of elements in the array within the variable `n`. In this example `n=5`. The outer `for` loop starting from line-12 counts the number of rounds required to sort the array. We have seen that for `n` elements we require a maximum of `(n-1)` rounds to sort the array. Hence this loop runs from `i=0` to `i<(n-1)` i.e. a total of `(n-1)` rounds.

The inner `for` loop, used for comparing elements of the array, starts from line-13. We have seen in round-1 `arr[0]` is compared with `arr[1]`, `arr[1]` is compared with `arr[2]` and so on, till `arr[3]` is compared with `arr[4]`. In doing so, at the end of round-1 the largest value occupies the last position i.e. `arr[4]`.

* (**niao**) = **n**ot **i**n **a**scending **o**rder; (**iao**) = **i**n **a**scending **o**rder

Therefore in round-2 there is no need to include the last element `arr[4]` for comparison and we are stopping at `arr[3]`. In general, **with each round we are comparing one element less from the end.** Hence the range of the counter variable `j` is taken from `j=0` to `j<n-i-1`. When `i=0`, we compare from `arr[0]` to `arr[5-0-1]` i.e. `arr[4]`. When `i=1`, we compare from `arr[0]` to `arr[5-1-1]` i.e. `arr[3]`. For `i=2`, we compare from `arr[0]` to `arr[5-2-1]` i.e. `arr[2]` and so on.

The `if` statement of line-14, is used to compare the element `arr[j]` with `arr[j+1]` for each value of `j`. In the above program we are arranging the values in ascending order. Therefore in case `arr[j]` and `arr[j+1]` are in ascending order, then no exchange takes place. However if the values are not in ascending order, then the values are exchanged, to put them in proper order. Finally the sorted array is printed using the `for` loop of line-20.

The next program is an alternative version of the **Selection Sort method.**



Alternative Selection Sort Method

```
1  /*Program-112: Alternative Selection Sort Method*/
2  #include<stdio.h>
3  #define MAX 100
4  void main()
5  {int arr[MAX], i, j, n, temp, pos, min;
6   printf("\nEnter total number of elements: ");
7   scanf("%d", &n);
8   for(i=0; i<n; i++)
9     {printf("\nEnter Element-%d: ", i);
10     scanf("%d", &arr[i]);
11    }
12  for(i=0; i<n-1; i++)
13    {min = arr[i];
14     pos = i;
15     for(j=i+1; j<n; j++)
16        {if( arr[j] < min )
17            {min = arr[j];
18             pos = j;
19            }
20        }
21     temp = arr[i];
22     arr[i] = arr[pos];
23     arr[pos] = temp;
24    }
25  for(i=0; i<n; i++)
26    printf("\nSorted array[%d] = %d", i, arr[i]);
27 }
```

Output:

```
Enter total number of elements: 8
Enter Element-0: 4
Enter Element-1: 1
Enter Element-2: 7
Enter Element-3: 9
Enter Element-4: 4
Enter Element-5: 3
Enter Element-6: 1
Enter Element-7: 8

Sorted array[0] = 1
Sorted array[1] = 1
Sorted array[2] = 3
Sorted array[3] = 4
Sorted array[4] = 4
Sorted array[5] = 7
Sorted array[6] = 8
Sorted array[7] = 9
```

In this method you start with the whole array and find the minimum value from the array. Then you exchange the minimum value with the value at the first position. So the first position now has the smallest value.

You next take the remaining part of the array i.e. from the second element to the last element and find the minimum value from this set. Finally exchange the second minimum value with the second element in the array. The second element therefore contains the second largest element now.

In this way you take each remaining section of the array and find the minimum in that section and exchange it with the first element of that set. When the process is over you finally have the array sorted.

The **for** loop in line-12 counts the number of sets. For an **n** element array there will be **(n-1)** sets (the first set has **n** elements, the second set has **n-1** elements, and in this way the last set will have **2** elements. There is no point in having a single element array for comparison). Hence the loop goes from **i=0** to **i=(n-2)**.

In line-13 the value at **arr[i]** i.e. **arr[0]** for the first set is stored in the variable **min**. The position **i** for the first value of the set is also stored in the variable **pos** in line-14.

The **for** loop of line-15 is used to compare the first value of the set at position **i** with the remaining values in the set. Hence for the set **i**, the loop starts from **j=i+1** and goes up to **j=n-1**.

Within the inner **for** loop, the **if** statement of line-16 checks if the value stored in position **j** is less than the value stored in **min**. If so, the new value replaces the value in **min**. The position **j** of the new **min** value is also stored in the variable **pos** in line-18.

The code in line-21 to 23 is then used to exchange the first value of the current set with the **min** value of that set as indicated by the position variable **pos**.

The **for** loop of line-25 is then used to print the sorted array.

The next program uses a function to delete multiple occurrences of a given value from the array.



Deleting multiple values from an array using function

```
1  /*Program-113: Deleting multiple values from an array*/
2  #include<stdio.h>
3  #define MAX 100
4  void del(int arr[], int k, int n)
5   { int i;
6     for( i=k; i<n-1; i++ )
7         arr[i] = arr[i+1];
8   }
9  void main()
10 {int arr[MAX], i, n, val;
11  printf("\nEnter total number of elements: "); scanf("%d", &n);
12  for(i=0; i<n; i++)
13     {printf("\nEnter Element-%d: ", i);
14      scanf("%d", &arr[i]);
15     }
16  printf("\nEnter value to delete: "); scanf("%d", &val);
17  i=0;
18  while(1)
19     { if(arr[i] == val)
20         {del(arr, i, n);
21          n--;
22         }
23       else
24          i++;
25       if(i==n) break;
26     }
27  printf("\nThe final array is: ");
28  for(i=0; i<n; i++)
29     printf("\n%d", arr[i]);
30 }
```

Output:

```
Enter total number of elements: 6
Enter Element-0: 2
Enter Element-1: 4
Enter Element-2: 2
Enter Element-3: 2
Enter Element-4: 6
Enter Element-5: 2
Enter value to delete: 2
The final array is:
4
6
```

The `del` function defined in line-4 deletes a value at position `k` from the array `arr` containing `n` elements. These are received by the function as formal arguments. Note the array notation used in the argument. The function is called by the `while` loop of line-18 every time an array element at position `i` is equal to the value `val` to delete. After a deletion, the array size `n` is reduced by 1 by the `n--` statement of line-21. After a deletion the next array element shifts to the position of the deleted value. As this value can also be equal to val, the array index `i` is not changed before checking it in the next iteration. Only when a value is not deleted, the array index `i` is incremented in line-24. As with each deletion the size `n` of the array gets reduced by 1, the `if` condition of line-25 checks if the new array index `i` exceeds the array size and `breaks` out.

The next program is used for doing a linear search. **Searching** is a technique that is used to **find a particular value from a list**. We will be discussing two methods, namely **liner search** and **binary search**.



Linear Search Method

```
1  /*Program-114: Searching for a value in an array using Linear Search Method*/
2  #include<stdio.h>
3  #define MAX 100
4  void main()
5  {int arr[MAX], i, n, num, flag=0;
6   printf("\nEnter total number of elements in the array: "); scanf("%d", &n);
7   for(i=0; i<n; i++)
8     {printf("\nEnter Element-%d: ", i);
9      scanf("%d", &arr[i]);
10    }
11  printf("\nEnter the number to search from the array: "); scanf("%d", &num);
12  for(i=0; i<n; i++)
13    {if(arr[i] == num)
14       {printf("\nValue present in position %d", i);
15        flag = 1; break;
16       }
17    }
18  if(flag == 0)
19    printf("\nValue not present in list");
20 }
```

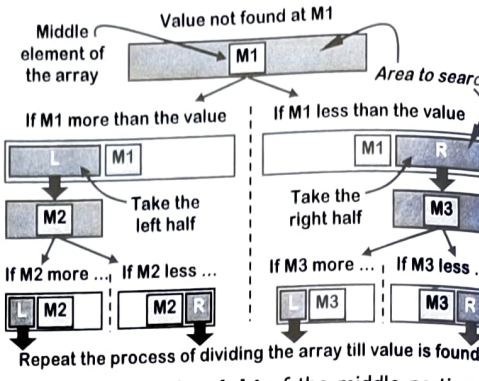
Output:

```
Enter total number of elements in the array: 5
Enter Element-0: 4
Enter Element-1: 7
Enter Element-2: 2
Enter Element-3: 9
Enter Element-4: 6
Enter the number to search from the array: 9
Value present in position 3
```

In program-114, we have declared an array called `arr[]` to store 100 integer values. In the example output shown, in line-6 we have input `5` in the variable `n` for storing 5 array elements. The values are entered using the `for` loop of line-7. In line-11 the value to search from the array is entered within the variable `num`. The `for` loop of line-12 is then used to compare the values stored in the array with the value in `num`. The `if` statement of line-13 is used to do the comparison. In case a match is found only then the value of the variable `flag` is changed to '1' and the control breaks out of the `for` loop. In case no match is found, the variable `flag` is not changed and remains at its initial value of '0'. In line-18, it is checked if `flag` is equal to '0'. If so then it prints that the value is not present in the list.

To use binary search, the values must be in a sorted order in an array.

The next program is used to **search for a value from a list using the binary search method**. This method is similar to finding a particular page of a book using its page number. But the **values must be present in either ascending or descending order** in the list.



In this method, to find a value, first the **value is searched at the middle of the array**. If the value is found at that position then it is displayed. Else two different conditions can arise. If the value at the middle of the array **(M1) is more** than the value to search, then take the half of the **array to the left** of the middle portion. If the value at the middle of the array **(M1) is less** than the value to search, then take the half of the **array to the right** of the middle portion. Next again search the value at the middle of this newly selected portion. This process of **dividing an array into two halves** and searching for the value at the middle portion of the selected half is continued till the number is finally found. The following program is used to do a binary search on a list of integer numbers. Three different cases can arise as indicated in the following examples.

Binary Search Method

```
1  /*Program-115: Search using Binary Search Method*/
2  #include<stdio.h>
3  #define MAX 100
4  void main()
5  {int arr[MAX], i, n, low, high, num, mid, flag=0;
6   printf("\nEnter total number of elements: ");
7   scanf("%d", &n);
8   for(i=0; i<n; i++)
9      {printf("\nEnter Element-%d: ", i);
10      scanf("%d", &arr[i]);
11     }
12  printf("\nEnter the number to search: ");
13  scanf("%d", &num);
14  low = 0;
15  high = n-1;
16  while(low<=high)
17     { mid = (low+high)/2;
18       if( arr[mid]==num )
19         {printf("\nPresent in position-%d", mid);
20          flag =1;
21          break;
22         }
23       if( arr[mid]>num )
24          high = mid-1;
25       else
26          low = mid+1;
27     }
28  if(flag == 0)
29     printf("\nValue not present in list");
30 }
```

Output:

```
Enter total number of elements: 6
Enter Element-0: 2
Enter Element-1: 3
Enter Element-2: 5
Enter Element-3: 6
Enter Element-4: 7
Enter Element-5: 9
Enter the number to search: 7
Present in position-4
```

An integer array `arr[]` is declared in line-4 to hold a maximum of 100 integers. In line-6 the actual number of values are entered into the variable `n`. For the output shown, `n=6` in our example. The `for` loop of line-8 then enters these values. In line-13 the value to search from the list is entered into variable `num`.

The variables `low` and `high` are used to indicate the ends of the current portion of the array being searched. Since, the whole array is considered for the first time search, therefore `low=0` and `high=n-1` (the index of the first and last elements in the list) as assigned in lines-14 and 15.

The `while` loop of line-16 is used to search the array for the value `num`. **Note the condition for the `while` loop.** In case the value `num` is present in the list, then during the search the value of `low` will be always less than or equal to `high`. In case the value `num` is not present in the list, then at one point of time the value in the variable `low` will become greater than the value in the variable `high`. Since no further search is required, the condition becomes false under such a case, and the `while` loop is terminated.

Within the `while` loop, in line-17 the index of the middle element is calculated by the expression `(low+high)/2` and stored in the variable `mid`.

The `if` statement of line-18 then checks if the value at the position `mid` i.e. `arr[mid]` is equal to `num` or not. If equal, then the search is over and the position of the number is displayed. The variable `flag` is assigned the value '1' and the program breaks out of the `while` loop.

If `arr[mid]` is not equal to `num`, then two cases can arise.

1. If `arr[mid]>num`, then there is no point in searching for the number after the position `mid`. Therefore the portion of the array to the left of `mid` is considered for the next search. Accordingly `high` is made equal to `mid-1` (the index immediately to the left of `mid`).
2. If `arr[mid]<num`, then there is no point in searching for the number before the position `mid`. Therefore the portion of the array to the right of `mid` is considered for the next search. Accordingly `low` is made equal to `mid+1` (the index immediately to the right of `mid`).

With the new values of `low` or `high`, the middle index `mid` for the new search portion is again calculated in line-17 and the process repeated till the value is found.

In case the value is not present in the list, then the variable `flag` remains at '0'. This is checked in line-28 and the message printed accordingly.

As an example let the list of 6 values be {2, 3, 5, 6, 7, 9}. To start with, `low=0`, and `high=(6-1)=5`.

Case-I: Let value to search be `num=7` (see diagram)

Now `low=0, high=5,` ∴ `mid=(0+5)/2 = 2`

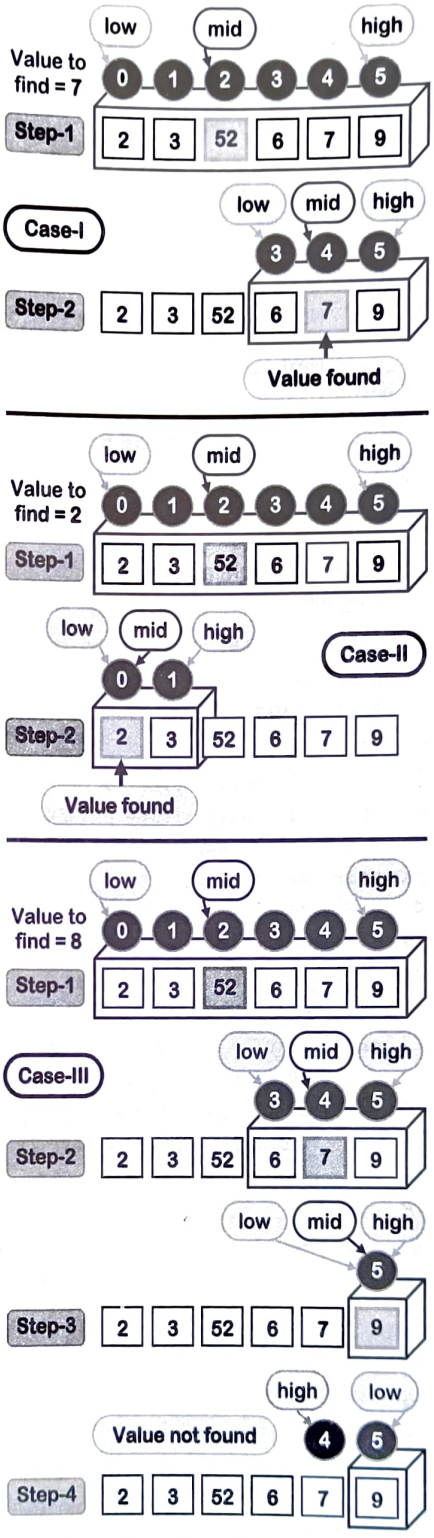
Hence `arr[mid] = arr[2] = 5;`

Now 5 < 7, therefore `low = mid+1 = 2+1 = 3`

Next `low=3, high=5,` ∴ `mid=(3+5)/2 = 4`

Hence `arr[mid] = arr[4] = 7;`

Now `num=7`, therefore value found at position-4



Case-II: Let value to search be `num=2` (see diagram)

Now `low=0, high=5,` ∴ `mid=(0+5)/2 = 2`

Hence `arr[mid] = arr[2] = 5;`

Now 5 > 2, therefore `high = mid-1 = 2-1 = 1`

Next `low=0, high=1,` ∴ `mid=(0+1)/2 = 0`

Hence `arr[mid] = arr[0] = 2;`

Now `num=2`, therefore value found at position-0

Case-III: Let value to search be `num=8` (see diagram)

Now `low=0, high=5,` ∴ `mid=(0+5)/2 = 2`

Hence `arr[mid] = arr[2] = 5;`

Now 5 < 8, therefore `low = mid+1 = 2+1 = 3`

Next `low=3, high=5,` ∴ `mid=(3+5)/2 = 4`

Hence `arr[mid] = arr[4] = 7;`

Now 7 < 8, therefore `low = mid+1 = 4+1 = 5`

Next `low=5, high=5,` ∴ `mid=(5+5)/2 = 5`

Hence `arr[mid] = arr[5] = 9;`

Now 9 > 8, therefore `high = mid-1 = 5-1 = 4`

For the next round, we have the condition `low=5` and `high=4`, therefore the condition is False and the control comes out of the loop indicating the value '8' is not present within the loop.

The following program is used to find the **transpose of a given matrix**. The transpose is obtained by changing the rows of the matrix as the columns of the matrix.



Transpose of a square matrix

```
/*Program-116: To find the transpose of a square matrix*/
#include<stdio.h>
#define MAX 10
void main()
{int mat[MAX][MAX], i, j, temp;
 printf("\nEnter total number of rows or columns in the matrix: ");
 scanf("%d", &n);
 for(i=0; i<n; i++)
   for(j=0; j<n; j++)
     {printf("\nEnter element-[%d][%d]: ", i, j);
      scanf("%d", &mat[i][j]);
     }
printf("\nThe original matrix is:\n");
 for(i=0; i<n; i++)
   {for(j=0; j<n; j++)
      printf("%d\t", mat[i][j]);
    printf("\n");
   }
 for(i=0; i<n; i++)
   for(j=0; j<n; j++)
     {if(i<j)
        {temp = mat[i][j];
         mat[i][j] = mat[j][i];
         mat[j][i] = temp;
        }
     }
 printf("\nThe required transpose of the matrix is:\n");
```

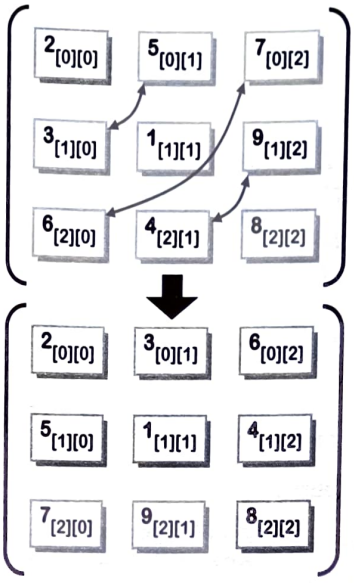
```
28    for(i=0; i<n; i++)
29      {for(j=0; j<n; j++)
30         {printf("%d\t", mat[i][j]);}
31       printf("\n");
32      }
33  }
```

Output:

```
Enter total number of rows or columns in the matrix: 3
Enter element-[0][0]: 2
Enter element-[0][1]: 5
Enter element-[0][2]: 7
Enter element-[1][0]: 3
Enter element-[1][1]: 1
Enter element-[1][2]: 9
Enter element-[2][0]: 6
Enter element-[2][1]: 4
Enter element-[2][2]: 8

The original matrix is:
2    5    7
3    1    9
6    4    8

The required transpose of the matrix is:
2    3    6
5    1    4
7    9    8
```



The transpose of a matrix can be obtained by **exchanging the values** in the matrix for unequal indices i.e. **where the index `i` is not equal to the index `j`**. This is evident from the diagram shown on the right. The first matrix gives the original matrix and the second matrix gives the transposed matrix.

The second element of the first row of the original matrix is 5 and its index is `[0,1]`. In the transposed matrix, the index of the value 5 is `[1,0]`. Similarly the index of the value 3 in the original matrix is `[1,0]`, while that in the transposed matrix is `[0,1]`. Thus it is seen that there is an exchange in the position of the values 5 and 3. Similarly, the other pairs of values i.e. $7_{0,2}$ & $6_{2,0}$ and $9_{1,2}$ & $4_{2,1}$ have changed their places. This change in place can be done by changing the index of these values. Thus the value $5_{0,1}$ is exchanged with the value $3_{1,0}$ to get the corresponding values in the transposed matrix.

Note that for the diagonal elements, there is no need to change the values, as the diagonal remains same in both the original and the transposed matrix.

Moreover the exchange should take place only once. Thus 5 is exchanged with 3 only once i.e. the element at `[0][1]` is exchanged with `[1][0]`, but element `[1][0]` is not again exchanged with `[0][1]`. If so, the values would be exchanged once more and we would get back the original matrix.

In line-5 of the program we have declared a square matrix `mat[][]` with 10 rows and 10 columns. In line-7, we have entered the actual number of rows and columns within the variable `n`. The `for` loops of line-8 and 9 are used to enter the values into the matrix. The outer loop with index `i` is used to traverse a row, while the inner loop with index `j` is used to traverse the elements in a particular row indicated by `i`. Thus line-11 enters a value into the variable `mat[i][j]`.

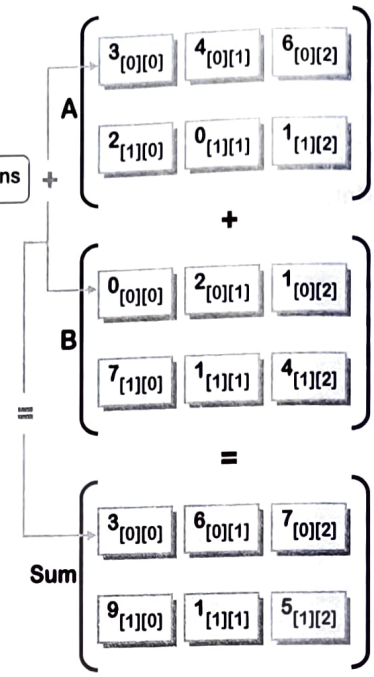
The `for` loops of lines-14 and 15 are used to print the input matrix. The `for` loops of lines-19 and 20 are used to create the transpose matrix. In line-21 it is checked if `i` and `j` are unequal i.e. if `i<j`. This ensures that we are dealing with only one exchange between `[i][j]` and `[j][i]`. Lines-22 to 24 do the exchange between `mat[i][j]` and `mat[j][i]`. Note that for all elements above the diagonal, the index `i<j`.

Finally the `for` loops of lines-28 and 29 are used to print the final transposed matrix. Note how the matrix is getting displayed. The tab character `\t` ensures that the numbers in a row are printed at equal gaps. The new-line character `\n` in line-31 is used to go to the next line for a new row.

Addition of two matrices

The next program is used to **add two matrices**.

```
1  /*Program-117: Addition of two matrices*/
2  #include<stdio.h>
3  void main()
4     { int A[2][3], B[2][3], Sum[2][3], i, j;
5       for(i=0; i<2; i++)
6          for(j=0; j<3; j++)
7             {printf("\nEnter value A[%d][%d]: ", i, j);
8              scanf("%d", &A[i][j]);
9              printf("\nEnter value B[%d][%d]: ", i, j);
10             scanf("%d", &B[i][j]);
11             Sum[i][j] = A[i][j] + B[i][j];
12            }
13      printf("\nThe required sum matrix is:\n");
14      for(i=0; i<2; i++)
15         {for(j=0; j<3; j++)
16            {printf("%d\t", Sum[i][j]);}
17          printf("\n");
18         }
19  }
```



Output

```
Enter value A[0][0]: 3
Enter value B[0][0]: 0
Enter value A[0][1]: 4
Enter value B[0][1]: 2
Enter value A[0][2]: 6
Enter value B[0][2]: 1
Enter value A[1][0]: 2
Enter value B[1][0]: 7
Enter value A[1][1]: 0
Enter value B[1][1]: 1
Enter value A[1][2]: 1
Enter value B[1][2]: 4

The required sum matrix is:
3       6       7
9       1       5
```

The above program is a simple one that is used to add two different matrices to get a sum matrix. Three rectangular matrices `A[][]`, `B[][]`, and `Sum[][]` are declared in line-4. Each has 2 rows and 3 columns. The `for` loops of lines-5 and 6 are used to enter the values into the matrices `A[][]` and `B[][]`. The `i` loop controls the rows, while the `j` loop controls the columns. Line-8 enters the element `A[i][j]`, while line-10 enters the value `B[i][j]` for every `i` and `j`.

Line-11 is then used to add the values `A[i][j]` and `B[i][j]` and assigns the result to the element `Sum[i][j]`. Note that for addition of two matrices, all the three matrices involved should be identical in size. The `for` loops of line-14 and 15 are used to print the `Sum[][]` matrix.

Checking a matrix with a row with all '0's

The next program is used to check if a particular **row of a matrix has all the elements as zeroes**.

```
1  /*Program-118: To check if a matrix has a row with all 0's*/
2  #include<stdio.h>
3  #define MAX 3
4  void main()
5     { int mat[MAX][MAX], i, j, count;
```

```
    for(i=0; i<MAX; i++)
      for(j=0; j<MAX; j++)
        {printf("\nEnter value mat[%d][%d]: ", i, j);
         scanf("%d", &mat[i][j]);
        }
    printf("\nThe input matrix is:\n");
    for(i=0; i<MAX; i++)
      {for(j=0; j<MAX; j++)
         printf("%d\t", mat[i][j]);
       printf("\n");
      }
    for(i=0; i<MAX; i++)
      { count=0;
        for(j=0; j<MAX; j++)
          {if(mat[i][j] == 0);
             count++;
          }
        if(count==MAX)
          printf("\nRow %d has all zeroes", i);
      }
}
```

Output

```
Enter value mat[0][0]: 2
Enter value mat[0][1]: 6
Enter value mat[0][2]: 7
Enter value mat[1][0]: 0
Enter value mat[1][1]: 0
Enter value mat[1][2]: 0
Enter value mat[2][0]: 8
Enter value mat[2][1]: 2
Enter value mat[2][2]: 4
The input matrix is:
2   6   7
0   0   0
8   2   4
Row 1 has all zeroes
```

Line-5 declares a square matrix `mat[][]` with 3 rows and 3 columns. The `for` loops of line-6 and 7 are then used to enter the values into the matrix. The `scanf()` of line-9 then enters the values. The `for` loops of lines-12 and 13 are used to print the input matrix.

The `for` loops of line-17 and 19 are then used to check for a row with all zeroes. In line-18, after the outer `for` loop, the variable `count` is initialised to `0`. This variable is used to count the number of zeroes in a given row. In this example, a row contains 3 elements or columns. Hence if all the elements in a row are '0', then `count` will be equal to 3 for that row.

The inner `j` loop is used to scan the columns for a particular row. The `if` statement of line-20 is used to check if `mat[i][j]` is equal to '0'. If so, then the variable `count` is incremented by 1. The `if` statement of line-23 then checks if `count` is equal to 3 or not. If so, then line-24 prints which row has all zeroes. The variable `count` is again initialised to '0' in line-18 for checking the next row.



Checking for an Identity Matrix

The next program is used to check if a particular matrix is an **Identity Matrix** or not.

```
/*Program-119: To check if a matrix is an Identity Matrix*/
#include<stdio.h>
#define MAX 3
int main()
  { int mat[MAX][MAX], i, j, flag=0;
    for(i=0; i<MAX; i++)
       for(j=0; j<MAX; j++)
          {printf("\nEnter value mat[%d][%d]: ", i, j);
           scanf("%d", &mat[i][j]);
          }
    printf("\nThe input matrix is:\n");
    for(i=0; i<MAX; i++)
       {for(j=0; j<MAX; j++)
          {printf("%d\t", mat[i][j]);}
        printf("\n");
       }
    for(i=0; i<MAX; i++)
       {for(j=0; j<MAX; j++)
            { if( (i!=j) && (mat[i][j]!=0) )
                {flag=1;
                 break;
                }
              if( (i==j) && (mat[i][j]!=1) )
                {flag=1;
                 break;
                }
            }
        if (flag==1) break;
       }
    if(flag==0)
       printf("\nMatrix is an Identity Matrix");
    else
       printf("\nMatrix is not an Identity Matrix");
    return 0;
  }
```

Output

```
Enter value mat[0][0]: 1
Enter value mat[0][1]: 0
Enter value mat[0][2]: 0
Enter value mat[1][0]: 0
Enter value mat[1][1]: 1
Enter value mat[1][2]: 0
Enter value mat[2][0]: 0
Enter value mat[2][1]: 0
Enter value mat[2][2]: 1

The input matrix is:
1    0    0
0    1    0
0    0    1

Matrix is an Identity Matrix
```

Line-5 declares a square matrix `mat[][]` with 3 rows and 3 columns. The variable `flag` declared in line-4 will be used later to check if the matrix is an identity matrix or not. The `for` loops of line-6 and 7 are then used to enter the values into the matrix. The `scanf()` of line-9 then enters the values. The `for` loops of lines-12 and 13 are used to print the input matrix.

The nested `for` loops of line-17 and 18 are then used to check the matrix to be an Identity matrix or not. For a matrix to be an Identity Matrix, **all non-diagonal elements should be equal to '0'**. To check non-diagonal element `i!=j` is checked. The condition that an element is non-diagonal but not equal to '0' is tested by the `if` statement of line-19 using logical AND operation to connect the two conditions.

In case we have a non-diagonal element which is not equal to '0', the condition of the `if` statement of line-19 becomes true. As there is no further need to check for the remaining matrix, the variable `flag` is made equal to '1' and the program breaks out of the inner `for` loop.

For a matrix to be an Identity Matrix the second condition to satisfy is, **all diagonal elements should be equal to '1'**. For a diagonal element `i==j` is checked. The condition whether an element is diagonal but not equal to '1' is tested by the `if` statement of line-23 using logical AND operation to connect the conditions.

In case a diagonal element is not equal to '1' there is no need to check any further and the variable `flag` is made equal to '1'. The control breaks out of the second inner `for` loop using a `break` statement in line-25.

In case the `break` statements of line-21 and 25 are run, the program comes out of the inner `for` loop of line-18. However it still remains inside the outer `for` loop. To come out of the outer `for` loop when `flag` becomes equal to 1, the `if` condition is checked and the `break` statement of line-28 is executed.

For an Identity Matrix, the conditions in lines-19 and 23 will never be true. Accordingly the value of the variable `flag` will remain at '0'. Hence the `if` statement of line-30 will print the matrix to be an identity matrix. On the other hand if the matrix is not an identity matrix, then either of the `if` statements of lines-19 or 23 will be true, and the variable `flag` will be changed to 1. The `if` condition of line-30 will check accordingly and the program will print that the matrix is not an identity matrix.

The next program is used to check if a particular matrix is an **Upper Diagonal Matrix** or not.

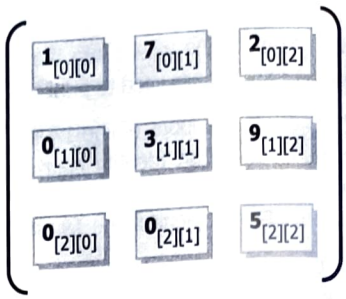


Check for Upper Diagonal Matrix

```
1  /*Program-120: To check if a matrix is an upper diagonal matrix*/
2  #include<stdio.h>
2  void main()
3    { int mat[3][3], i, j, flag=0;
4      for(i=0; i<3; i++)
5        for(j=0; j<3; j++)
6          {printf("\nEnter value mat[%d][%d]: ", i, j);
7           scanf("%d", &mat[i][j]);
8          }
9      printf("\nThe input matrix is:\n");
10     for(i=0; i<3; i++)
11       {for(j=0; j<3; j++)
12          printf("%d\t", mat[i][j]);
13        printf("\n");
14       }
15     for(i=0; i<3; i++)
16        for(j=0; j<3; j++)
17           {if( (i>j) && (mat[i][j]!=0) )
18               {flag=1;
19                break;
20               }
21           }
22     if(flag==0)
23        printf("\nMatrix is an Upper Diagonal Matrix");
24     else
25        printf("\nMatrix is not an Upper Diagonal Matrix");
26   }
```

Output

```
Enter value mat[0][0]:  1
Enter value mat[0][1]:  7
... ... ...
Enter value mat[2][1]:  0
Enter value mat[2][2]:  5
The input matrix is:
1    7    2
0    3    9
0    0    5
Matrix is an Upper Diagonal Matrix
```



The program is similar to the previous one. The difference lies in the `for` loops of lines-16 and 17. An upper diagonal matrix implies that the matrix has values in its diagonal and above the diagonal. All elements below the diagonal are '0'. To test for elements below the diagonal, note that such elements have an index such that `i>j`. This is checked by the `if` statement of line-18. If the condition is satisfied, i.e. if the element is below-diagonal and it is not equal to 0, it makes the variable `flag` as 1 and breaks out of the loop in line-19.

The next program is used to **multiply two matrices**.

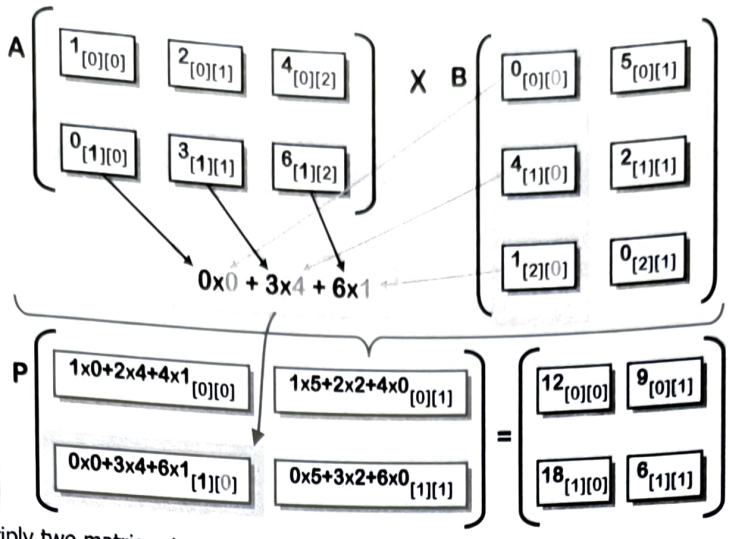


Program for Matrix Multiplication

```
1  /*Program-121: Matrix Multiplication*/
2  #include<stdio.h>
3  #define MAX 3
4  void main()
5    { int A[MAX][MAX], B[MAX][MAX], P[MAX][MAX]={0}, i, j, k;
6      for(i=0; i<MAX; i++)
7         for(j=0; j<MAX; j++)
8            {printf("\nEnter value A[%d][%d]: ", i, j);
9             scanf("%d", &A[i][j]);
10            printf("\nEnter value B[%d][%d]: ", i, j);
11            scanf("%d", &B[i][j]);
12           }
13     printf("\nThe input matrix A:\n");
14     for(i=0; i<MAX; i++)
15        {for(j=0; j<MAX; j++)
16           printf("%d\t", A[i][j]);
17         printf("\n");
18        }
19     printf("\nThe input matrix B:\n");
20     for(i=0; i<MAX; i++)
21        {for(j=0; j<MAX; j++)
22           printf("%d\t", B[i][j]);
23         printf("\n");
24        }
25     for(i=0; i<MAX; i++)
26         for(j=0; j<MAX; j++)
27             for(k=0; k<MAX; k++)
28                 P[i][j] = P[i][j] + A[i][k] * B[k][j];
29     printf("\nThe product matrix is:\n");
30     for(i=0; i<MAX; i++)
31        {for(j=0; j<MAX; j++)
32             printf("%d\t", P[i][j]);
33         printf("\n");
34        }
35   }
```

Output

```
Enter value A[0][0]:  1
Enter value B[0][0]:  2
Enter value A[0][1]:  3
Enter value B[0][1]:  6
... ... ...
Enter value A[2][2]:  5
Enter value B[2][2]:  7
The input matrix A:
1    3    0
2    0    4
3    1    5
The input matrix B:
2    6    4
0    1    2
3    2    7
The product matrix is:
2    9    10
16   20   36
21   29   49
```



Process of Matrix Multiplication

The above program is used to multiply two matrices to get a resultant matrix.

Any two matrices cannot be multiplied. For two matrices to be compatible for multiplication, **the number of columns of the first matrix should be equal to the number of rows of the second matrix**.

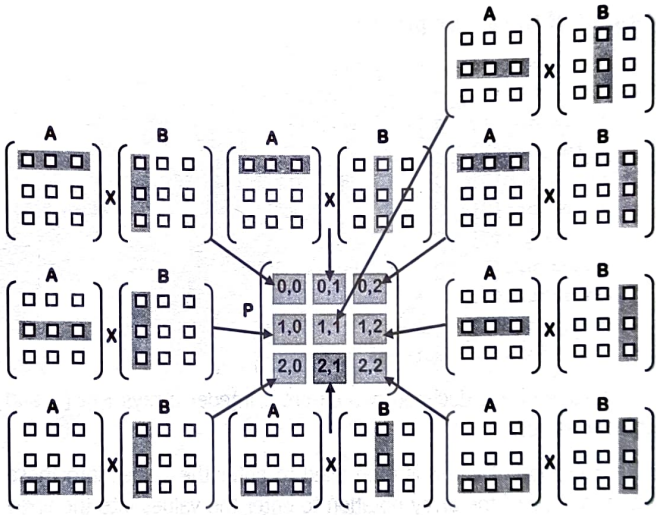
Thus for the two matrices `A[m][n]` and `B[n][q]`, the product matrix `P[][]` will be:

`A[m][n] x B[n][q] = P[m][q]`
(`n` same)

For matrix multiplication the number of columns of the first matrix should be equal to the number of rows of the second matrix.

Note that for `A[][]`, the number of columns is '`n`' and for `B[][]` the number of rows is also '`n`'. Also note that the product matrix has the number of rows of `A` and the number of columns of `B`.

To calculate an element of the product matrix we have to carry out the following steps. Let us take the example shown above where `A[2][3]` (m=2, n=3) is multiplied with `B[3][2]` (n=3, q=2) to get the product matrix `P[2][2]`.



As a general rule note that the element `P[i][j]` is obtained by multiplying the ith row of `A[][]` with the jth column of `B[][]`. The diagram on the right gives a visual idea of which row and which column of the matrices `A` and `B` get multiplied to get a particular element of the product matrix `P`. Thus to get the value of the element `P[1][0]` for example, we have to multiply values from row-1 of matrix `A[][]` with values from column-0 of matrix `B[][]`. In doing so, we have to multiply each element of row-1 with each element of column-0 and add the terms, as shown below:

The element `P[1][0] = A[1][0]*B[0][0] + A[1][1]*B[1][0] + A[1][2]*B[2][0]`

In general `P[i][j] = ΣA[i][k]*B[k][j]`, where the summation Σ is made over k (here k is the number of columns of `A[][]` or the number of rows of `B[][]`).

In program 121, in line-5 three different square matrices `A`, `B`, and `P` have been declared to store 3x3 elements each. Note that the product matrix `P[3][3]` has been initialised to '0' in that line. The reason is that each element of `P` will be a sum and is of the form `sum=sum+term`. Hence each element needs to be initialised to '0'.

The `for` loops of lines-6 and 7 are used to enter the values into the matrices `A` and `B`. The `for` loops of lines 14, 15 and lines 20, 21 are then used to print the two input matrices `A` and `B`.

Lines-25 to 28 are used to calculate the product matrix. The `for` loop of line-27 is used to do the summation of the terms, for a particular element `P[i][j]` of the product matrix. The summation is done over index `k`.

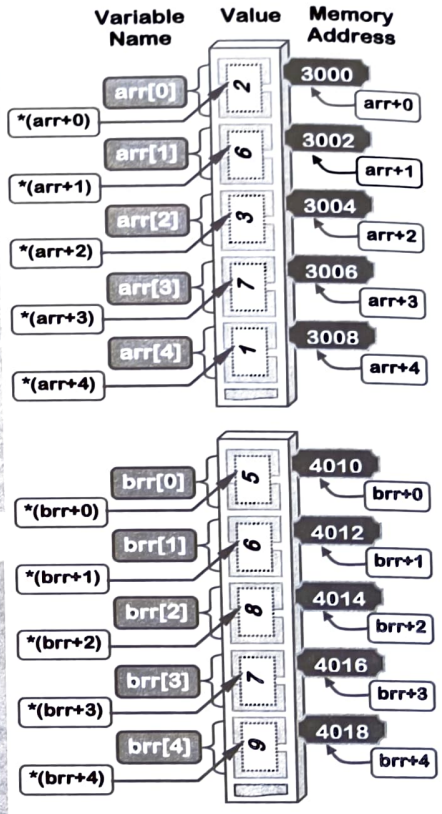
Finally the `for` loops of line-30 and 31 are used to display the final product matrix.

The next example **compares two different arrays** and finds out at how many places the values are same in those arrays using **pointer notation**.



Comparing arrays using pointers

```
1   /*Program-122:Comparing 2 arrays using pointers*/
2   #include<stdio.h>
3   #define MAX 5
4   void main()
5     { int arr[MAX], brr[MAX], i, count=0;
6       for(i=0; i<MAX; i++)
7           {printf("\nEnter arr[%d]: ", i);
8            scanf("%d", arr+i);        /*Address of arr[i]*/
9            printf("\nEnter brr[%d]: ", i);
10           scanf("%d", brr+i);        /*Address of brr[i]*/
11          }
12      for(i=0; i<MAX; i++)
13         {if(*(arr+i) == *(brr+i)) /*Elements compared*/
14            count++;
15         }
16      printf("\nThe number of matches = %d", count);
17    }
```



Output of the above program:

```
Enter arr[0]: 2
Enter brr[0]: 5
Enter arr[1]: 6
Enter brr[1]: 6
Enter arr[2]: 3
Enter brr[2]: 8
Enter arr[3]: 7
Enter brr[3]: 7
Enter arr[4]: 1
Enter brr[4]: 9
The number of matches = 2
```

In line-5 we have declared two different integer arrays `arr[]` and `brr[]` each containing a maximum of 5 integer values.

The `for` loop of line-6 is then used to enter the values into these arrays. Note how the pointer notation is used instead of the array notation to enter the values into the array using the `scanf()` statements of 8 and 10. Thus `arr+i` and `brr+i` are written instead of writing `&arr[i]` and `&brr[i]`.

The `for` loop of line-12 is then used to access the array elements and compare them. The `if` structure of line-13 compares the two arrays, again using pointer notation. Thus instead of using `arr[i]` and `brr[i]`, we have used the indirection operator to dereference the address of the array elements to get the values. Hence the notations `*(arr+i)` and `*(brr+i)` are used.

Whenever there is a match in values in the arrays `arr` and `brr` at a particular position, the variable `count` is incremented by 1 using the statement of line-14. In our example, the two arrays contain same values at positions `i=1` and `i=3`. Hence the `count` value gets incremented when `i=1` and `i=3`.

Finally in line-16, the `count` value is printed.

C to it that .....

The following examples demonstrate the areas where mistakes are common. The whole program is not written always and only the relevant parts are shown.

Look at the following piece of code.

```
1  void main()
2  { int n, arr[n], i;
3    printf("\nEnter number of elements:");
4    scanf("%d", &n);
5    for(i=0; i<n; i++)
6      {printf("\nEnter value-%d: ", i);
7      scanf("%d", &arr[i]);
8      }
9    for(i=0; i<n; i++)
10     printf("\n%d: ", arr[i]);
11 }
```

When the following code is compiled, it will compile. But when the above code is run, it will stop working and give an error message. The reason is the wrong way of declaring an array as in line-2. An array cannot be declared with a variable number of elements like `arr[n]` where `n` is input later as in line-4.

The problem can be rectified in two different ways. In the first case assign a value to `n` before the array is declared as shown below. Also remove the `scanf()` operation to enter the number of elements.

```
1  void main()
2  {int n=5, arr[n], i;
3    for(i=0; i<n; i++)
4      {printf("\nEnter value-%d: ", i);
5       scanf("%d", &arr[i]);
6      }
7    for(i=0; i<n; i++)
8      printf("\n%d: ", arr[i]);
9  }
```

In the second case declare an array with a predefined number of elements. Next input a value of `n` as per the requirement using a `scanf()` operation as shown below.

```
1  void main()
2  {int n, arr[100], i;
3    printf("\nEnter number of elements:");
4    scanf("%d", &n);
5    for(i=0; i<n; i++)
6      {printf("\nEnter value-%d: ", i);
7       scanf("%d", &arr[i]);
8      }
9    for(i=0; i<n; i++)
10     printf("\n%d: ", arr[i]);
11 }
```

The following program shows a logical error that can occur while using arrays and the **while loop**.

```
1  void main()
2  { int arr[10], i=0, sum=0;
3    float avg;
4    for(i=0; i<10; i++)
5      {printf("\nEnter value-%d: ", i);
6       scanf("%d", &arr[i]);
7      }
```

```
8     i=0;
9     while(i++<10)
10       sum = sum + arr[i];
11    avg = sum/10.0;
12    printf("\nThe required average is %f", avg);
13 }
```

Output:

```
Enter value-0: 1
Enter value-1: 2
Enter value-2: 3
Enter value-3: 4
Enter value-4: 5
Enter value-5: 6
Enter value-6: 7
Enter value-7: 8
Enter value-8: 9
Enter value-9: 10
The required average is -2879.800049
```

The **for** loop of line-4 is used to enter the values in the array **arr[]** from **i=0** to **i=9**. The loop counter **i** is again initialised to '0' in line-8. The **while** loop of line-9 is then used to calculate the **sum**. The average is obtained by dividing the **sum** by 10. The final result is displayed in line-12.

However the output is not as expected. Ideally the output should be equal to 5.5. However it shows a value of -2879.800049. The reason lies in the wrong use of the **while** loop in line-9. Though the condition of the **while** loop is correct i.e. the loop will run from **i=0** to **i=9**, with the post increment operator ++ increasing the value of **i** by 1 after checking the condition. Thus the loop runs for a total of 0 to 9 i.e. 10 times.

The flaw lies in the use of the increment operator **i++**. When **i=0**, the condition of **while** checks the condition as **0<10** (which is true) and the **i++** statement then increments **i** to 1. Thus when the **sum** is made in line-10, it uses the array element **arr[1]** and not **arr[0]**. Similarly when the last condition is checked for **i=9** and **9<10**, then **i++** increments **i** to 10. When the **sum** is made, it adds **arr[10]** and not **arr[9]**. However **arr[10]** does not exist and the memory location there contains garbage value. When this garbage value is added to **sum**, it produces this absurd result.

The code should be rectified in the following manner to get the proper result.

```
8     i=0;
9     while(i++<10)
10       sum = sum + arr[i-1];
```

The following code that involves **array initialisation** will **neither compile nor run**:

```
1  void main()
2  { int arr[5], i;
3    arr = {4,9,6,3};
4    for(i=0; i<n; i++)
5       printf("\n%d: ", arr[i]);
6  }
```

The fault lies in the **incorrect initialisation of the array** in line-3. An array can be initialised with values only during its declaration as shown below:

```
2    int arr[5] = {4,9,6,3}, i;
```

The next code will show a compilation error as you **cannot leave space for values** to be entered later into a semi-initialised array.

```
1    int arr[6] = {4,,6,3,,7}, i;
```

You must use values to reserve space during array initialisation as shown below:

```
1    int arr[6] = {4,0,6,3,0,7}, i;
```

The Fact File

- **An array is a collection of similar type of variables under a common variable heading or name**
- All elements of any given array must be of the same data type, i.e. we cannot have an array of numbers some of which are int and some float etc. Either all should be int or all should be float
- **The different components of an array are called the array elements**
- In the declaration "int physics[20];" first, the set of square brackets [ ] tells the compiler that we are not dealing with a simple variable but an array. This example declares an array named physics (just like other variable names) which can store 20 int type numbers
- **The numbering of array elements starts from 0 and not from 1, i.e. an array with 100 elements will have the elements numbered from '0' to '99' and not from '1' to '100'. In general array[n] will have data elements from array[0] to array[n-1]**
- The array elements of a particular array are stored in consecutive memory locations depending upon the type of data
- **There are two methods to enter data into an array. These are by initialising an array with data elements during design time, or by entering data from the keyboard during run time**
- A for-loop is used in general to input data into an array (though the while and do-while loops can also be used if required)
- **While entering data into an array, care should be taken so that the input data does not surpass the number of elements declared**
- An array can be initialised by writing down the initial data elements that the array can hold. (An un-initialised array contains garbage data, unless it has been declared as a static variable array). The method to do this is to write down the array elements separated by commas within curly braces, as: int arr[5]={ 2, 4, 7, 4, 9 };
- **One can initialise an array only during the declaration of the array**
- In case the number of elements of an array is specified within the square brackets and all the elements are not initialised, then the un-initialised elements will all be initialised to 0 automatically
- **To initialise all the ten elements of the array arr[10] to 0 one can simply write:**
  int arr[10] = { 0 };
- In general one requires a maximum of (n-1) traversals of the array to sort an array with n number of elements using Selection Sort or Bubble Sort techniques
- **In Selection Sort, the smallest element in the array occupies the first position in the array after the first round of sorting**
- In Bubble Sort, the largest element occupies the last position in the array after the first round of sorting
- **Searching is a technique that is used to find a particular value from a list of values. There are various methods for searching like liner search technique and binary search technique**
- Binary Search Method is similar to finding a particular page of a book using its page number
- **The array elements must be in a sorted order to apply Binary Search**
- In Binary Search, the process of dividing an array into two halves and searching for the value at the middle portion of the selected half is continued till the number is finally found
- **A matrix is a 2 dimensional array with m rows and n columns. The row number and column number together identify any element in the matrix. Thus int mat[4][3] defines a matrix with 4 rows and 3 columns**
- The numbering of each row starts from '0' and extends up to one less than the total number of rows. Similarly the numbering of each column starts from '0' and extends up to one less than the total number of columns. The combination of a row number and a column number gives the position of any number in the matrix
- **The general rule is that arrays are stored in memory such that the rightmost index value varies most rapidly**
- In the row major form of storing the values of a matrix the elements are stored row wise. In this method after storing the elements of a row in consecutive locations, the elements of the next row are stored.
- **In the column major form of storing the values of a matrix the elements are stored column wise. In this method after storing the elements of a column in consecutive locations, the elements of the next column are stored**
- For two matrices to be compatible for multiplication, the number of columns of the first matrix should be equal to the number of rows of the second matrix
- **C provides us with a special type of variable called a 'pointer' to store the memory address locations of other variables**

- **A special operator called the 'address of' operator represented by the ampersand symbol (&) is used to access the address of a variable**
- Declare a pointer variable by first stating the data type of the pointer, followed by a ' * ', followed by the pointer name. The data type of that variable whose address it will store, will form the data type of the pointer
- **The declaration int *px indicates that px is a pointer type variable that can store the address of an integer type variable**
- A special operator called the indirection operator (*) is used to access the value of a variable using its address
- **The statement printf("Value of x=%d", *ptr); uses a pointer to display the value stored at the memory address indicated by the pointer ptr.**
- The indirection operator can also be called the value-pointed-by-address operator or value-stored-at-address operator as it can be used to display the value stored at a particular address location using that address location
- **The name of a one-dimensional array gets automatically converted to a pointer and the array name points to the starting element of the array**
- The name of an array can be treated as a pointer
- **Pointers increment in units depending on the data type they point to**
- For an array, when an integer i is added to the array name, the resultant expression points to the ith array element. In general (arr+i) ⇒ &arr[i] ⇒ address of the ith element of the array arr
- ***(arr + i) is same as arr[i] for the array arr**
- *arr indicates the value stored at the 0th position of the array arr
- **An integer can be added to a pointer or subtracted from a pointer**
- Two pointers can be compared
- **Two pointers cannot be added, multiplied, or divided**
- Two pointers can be subtracted
- **To subtract one pointer from another it needs to be assured that both the pointers point to elements of the same array or memory block**

Review Questions

Q1. **Multiple Choice Questions. Select from any one of the four options.** **1 each**

i) Which of the following is an incorrect statement for an array:
   a. All the array elements of a given array are of the same data type
   b. The array index starts from 0
   c. The name of an array indicates the value of the first element of the array
   d. Array elements of a given array store consecutive locations in memory

ii) Which of the following is an incorrect statement for an array:
   a. For an n element array the last element index is n-1
   b. You can change the size of an array as per requirement
   c. A numeric array declared using the static keyword has all its elements initialised to zero
   d. An array is passed to a function as argument by using the array name only

iii) For the statement int arr[8]={2, 6, 3, 9}; the array element arr[7] will store the value:
   a. garbage value b. 2 c. 9 d. 0

iv) For the statement int arr[8]={2, 6, 3, 9}; the array element arr[8] will store the value:
   a. 2 b. 0 c. garbage value d. 9

v) In bubble sort method:
   a. The last element of the array gets sorted first
   b. The first element of the array gets sorted first
   c. The second element of the array gets sorted first
   d. The second last element of the array gets sorted first

vi) Which of the following is an invalid code?
   a. int a[7]; b. int b[ ]; c. int c[ ] = {2, 8} d. int d[10] = {0};

vii) Which of the following is a valid statement for an array declared as int a[10];?
   a. *(a+2)=5; b. a=*(a+1); c. a*=(a+1); d. *a=a+1;

viii) In selection sort method:
a. The second element of the array gets sorted first
b. The last element of the array gets sorted first
c. The first element of the array gets sorted first
d. The second last element of the array gets sorted first

ix) For an array with n elements the maximum number of times the array needs to be scanned for sorting it using the bubble sort method is:
a. n b. n-1 c. n-2 d. n-3

x) For binary search method which of the following is not true:
a. The middle element of the array is searched first
b. The search area of the array gets divided into two portions after each unsuccessful search
c. The entire array needs to be in a sorted order
d. The algorithm works only if the search value is present in the array

xi) A pointer is:
a. A special constant that stores the address of another pointer
b. A special constant that stores the address of another variable
c. A special variable that stores the address of another pointer
d. A special variable that stores the address of another variable

xii) Which symbol is used to declare a pointer?
a. + b. * c. ⊕ d. x

xiii) What is the output of the following program piece in C?

```
int a[9]={3,4,5}, i, sum=0;
for(i=1; i<9; i++) sum+=a[i];
printf("\n%d", sum);
```

a. 5 b. 12 c. 9 d. 10

xiv) What is the output of the following program piece in C?

```
int b[5]={1,2,3,4,5}, i, f=1;
for(i=0; i<4; i++); f=f*a[i];
printf("\n%d", f);
```

a. 1 b. 24 c. 5 d. 0

xv) What is the output of the following program piece in C?

```
int arr[6]={1,0,3,0,5}, i, count=0;
for(i=0; i<6; i++)
  if( arr[i] == 0) count++;
printf("\n%d", count);
```

a. 1 b. 2 c. 3 d. 4

xvi) What is the output of the following program piece in C?

```
int arr[10] = {9,8,7,6,5,4,3,2,1,0}, i;
for(i=1; i<10; i++)
  *(arr+i) = *(arr+i-1);
printf("\n%d", arr[9]);
```

a. 1 b. 0 c. 9 d. 2

xvii) What is the output of the following program piece in C?

```
int arr[ ]={6,6,6,6,6,6}, i;
for(i=5; i>-1; i--)
  *arr = *(arr+i)+1;
printf("\n%d", arr[0]);
```

a. 12 b. 8 c. 7 d. 11

xviii) The declaration int m[4][8] defines a matrix with:
a. 4 columns b. 4 rows c. 8 rows d. none of these

xix) Which of the following statements is incorrect?
a. int a[3]={0}; b. int a[5]={2, , 5, , 6}; c. int a[ ]={2,2,2}; d. int a[ ]={0};

xx) Which of the operations is not possible on a pointer in C?
a. Add an integer to a pointer b. Subtract two pointers c. Add two pointers d. Compare pointers

Q2. **Short Answer type questions:** **1 each**

i) What is an array?
ii) What is the range of indices for the elements of an array int arr[5].
iii) What does the name of an array specify?
iv) How can you initialise an integer array during declaration with the values 3, 6, 9?
v) What does the declaration float mat[2][5]; mean?
vi) What is the alternative notation for the notation *(arr+i) for a float type array arr?
vii) What is the simplest method of initialising all the elements of an integer array arr to 0?
viii) What do you mean by the column major method of storing elements in a matrix?
ix) What do you mean by the row major method of storing elements in a matrix?
x) State one difference between selection sort and bubble sort.
xi) State one difference between linear search and binary search.
xii) State any two valid operations that can be done on a pointer in C.

Q3. **Long Answer type questions:** **7 each**

i) Write a program to reverse the contents of an array. Explain the terms row major and column major matrices. What is the use of the indirection operator in C? 4+2+1

ii) Write a program to sort the contents of an integer array in descending order. What is the relation between a pointer and an array? State one difference between linear and binary search. 4+2+1

iii) Write a program to search for a value from a given array. State any two characteristics of an array type data structure. State one disadvantage of using an array to store a set of values. 4+2+1

Q4. **Assignment Programs:** **4 each**

i) Write a program to get the sum of the squares of the values stored in a 10 element array.
ii) Write a program to get the sum of the values stored at the even positions of the array.
iii) Write a program to get the sum of those values in an n element array which are multiples of 7.
iv) Write a program to print the contents of an array from the last to the first.
v) Write a program to find the average of values in an array and then print those values from the array which are more than the average value.
vi) Write a program to find the second maximum value from an array having n integers.
vii) Write a program to find the second minimum value from an array having n integers.
viii) Write a program to find the standard deviation of those values in an array which are multiples of 3.
ix) Write a program to delete the middle element of an array having an odd number of elements.
x) Write a program to delete all odd values from an array with the help of a delete function.
xi) Write a program to insert the value -1 after every occurrence of the value 5 in an array.
xii) Write a program to insert the value -1 before every occurrence of the value 5 in an array.
xiii) Write a C program to sort the values at the even positions in an array having n elements.
xiv) Write a C program to sort the values at the odd positions in an array having n elements.
xv) Write a program to sort the first half of an array in ascending order and the second half of the array in descending order for an array having 2n number of integer values.
xvi) Write a program to find whether an array contains any consecutive sequence of values (for example the array [2, 9, 4, 5, 6, 1, 3, 8, 9] has two consecutive sequences 4, 5, 6 and 8, 9).
xvii) Write a program to separately find the sum of left and right diagonal elements of a square matrix.
xviii) Write a program to check if any column in a matrix is having all zeroes.
xix) Write a program to separately find the sum of each column of a matrix and display them.
xx) Write a program to get the product of a matrix and its transpose for a square matrix.
xxi) Write a program to find the maximum from each row of a matrix and print each maximum.
xxii) Write a program to check if a matrix is orthogonal in nature (a square matrix is orthogonal if the product of the matrix and its transpose is an identity matrix. An identity matrix is itself orthogonal).

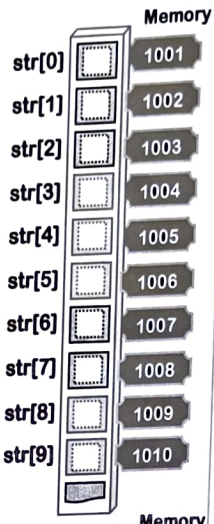
CHAPTER 14
Strings in C

- What are Strings? 14-1
- Entering and displaying Strings 14-2
- Matrix and Strings 14-4
- Some Library Functions available to handle Strings 14-5
- Pointers and Strings 14-10
- Some worked out problems 14-12

14.1 What are Strings?

Up to now we have written programs that had dealt with numerical values like integers and floating point numbers or at the most only simple character type data. But apart from these there exist another form of data called strings. By **string** we do not mean any wire or thread but a **collection of characters to form a word, name, sentence, paragraph etc.** However C does not provide us with any special data type to represent strings. But an **array of characters** has a special place in C. It is used to **denote a string**. A string is treated in C as a collection of characters in a character array, with a special marker to denote the end of a particular string. Thus the following line of code can be used to declare a string:

```
char str[10];
```



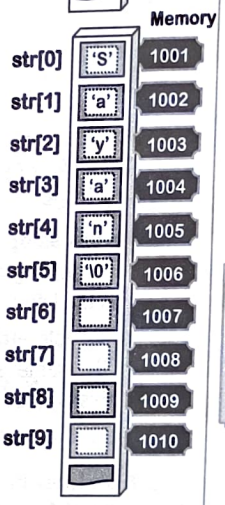
String

A String is a collection of characters to form a word, name, sentence, paragraph etc.

The memory diagram to the right shows how memory will be allocated for storing the above character array called `str` (the array can have any other name). Since a character occupies 1 byte, thus 1 byte will be allocated for storing each element of this array as shown. It is seen in this example that `str[0]` starts from memory location 1001 and the array ends at the memory location 1010 with the array element `str[9]`.

Just like initialising an integer or a float array, a character array can also be initialised to hold a string, as shown below. Each character is **enclosed within a pair of single quotes** and separated by a comma.

```
char str[10]={'S','a','y','a','n','\0'};
```



Initialising a String

A String is represented in C as a character array.

Notice that an **extra special character** '\0' has been introduced at the end of the name `Sayan` during the initialisation. The character '\0' is treated as a single character called a '**null indicator**' and is **used to mark the end of a string**.

The 'null indicator' i.e. **'\0' should be placed at the end of the set of characters** to tell C that a particular string, consisting of several characters, has ended. Without the null indicator it would not be possible to find the end of the string or to demarcate two or more consecutive strings. One important thing should be kept in mind. While declaring a character array for holding strings, **always remember to keep one character extra to include the null terminator**. Thus if a string consists of `n` characters, always declare the array to consist of `n+1` characters minimum, as `str[n+1]` and not as `str[n]`. In the latter case the character set will not be treated as a string as there would be no space to accommodate the null indicator.

Since an array of characters has a special place in C, it is not surprising that it can be also **initialised in a more convenient manner** as shown below:

```
char str[10]= "Sayan";
```

A String can be initialised only during declaration of the string array

The above declaration with the name `Sayan` between two double quotes " " is used to initialise the character array with a string. The **double quote indicates that this character array consists of a string** and C automatically inserts the null indicator '\0' after the last character in the string.

Remember that if you initialise a character array using single quotes as 'a', then the character is treated as a `char` type variable and if you initialise it using double quotes as in "a" then it is treated as a string and the **null indicator is automatically inserted at the end of the character**.

As discussed in the previous chapter, unlike other type of variables, an array can be initialised only during the declaration. **C does not allow directly assigning one array to another. Thus the code shown below is not allowed in C.**

```
char str[20];
str = "Sayan";
```

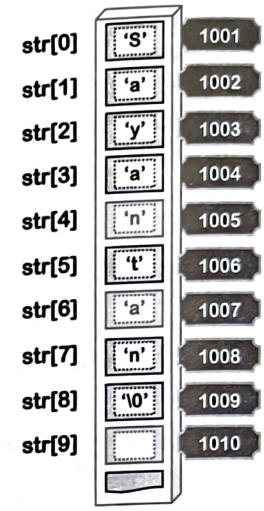
To assign a constant string (like the one shown above) to an array, special **functions like `strcpy()` are required** in C, as will be shown later.

14.2 Entering and displaying Strings

Apart from initialising, a string can also be input by using library functions like `scanf()`, `gets()` or `getchar()`. Similarly we can display a string using the `printf()` function and the `%s` format specifier or by using the `puts()` function. The following program is a simple one that inputs a word using the `scanf()` function and the `%s` format specifier and determines the number of letters in the word.

Program to find length of a string

```
1  /*Program-123: To find the length of a string*/
2  #include<stdio.h>
3  int main()
4    { char str[10];
5      int len;
6      printf("\nEnter the word to find its length: ");
7      scanf("%s", str);
8      for( len=0; str[len]!='\0'; len++);
9      printf("\nThe number of letters in the word = %d", len);
10     return 0;
11   }
```



Output

```
Enter the word to find its length: Sayantan
The number of letters in the word = 8
```

A special character called the null character '\0' is used to mark the end of the string.

Several points need to be noted in the simple program given above. In line-4 of the above program, the character array `str[10]` has been declared. The variable `len` declared in line-5 will be used to store the calculated length of the string. Several things need to be noted here:

- **No address operator '`&`' has been used in the `scanf()` function** before the variable name in line-7. When inputting a string using `scanf()` and the `%s` specifier there is no need to use the address operator '`&`', as by default the name the array indicates the address of the first element of the array i.e. &str[0].
- The variable array `str[]` inside `scanf()` function **does not include the square brackets `[]`** and only the name `str` is used. This is due to the same reason as above i.e. the name itself means &str[0] and this is what is required by the `scanf()` function as its second argument.
- Thirdly, the `%s` specifier **automatically inserts the null indicator** '`\0`' after the entered string.

Do not use the address operator '&' when inputting a string using scanf( )

The **`for` loop** in line-8 in the above program is used without any loop body following it and has been **terminated by a semicolon**. It may seem strange at first sight. The variable `len` is used to store the number of letters in the word. Within the `for` loop first `len` has been initialised to `0`. After the initialisation, the condition of the `for` loop checks the content of the string at `str[i]`.

Initially as the value of `len` is `0` the condition checks the first character of the word i.e. `str[0]`. In our example we have input the word "`Sayantan`". Therefore the condition `str[len]!='\0'` **is true** as `str[0]` **is equal to** '`S`' in our example. Since there is no statement as part of the loop, after the condition becomes true the control comes back to the increment part of the loop i.e. `len++` and `len` gets incremented by 1.

The condition subsequently checks for the consecutive values of the word with every increment of `len` and checks the characters `str[1]`, `str[2]`, `str[3]` etc. Any character other than '`\0`' will make the `for` loop condition true and let it continue. When the end of the string is reached, `len` will be equal to 8, and the last array element checked will be `str[8]` which is equal to '`\0`'. This will finally make the `for` loop condition

false and terminate the loop. Thus the final value of `len` will contain the total number of letters in the string. The required length stored in the variable `len` is finally displayed using the `printf()` function in line-9.

The `%s` specifier is perfect as long as we enter a single word. But if we enter multiple words separated by blanks and use the %s specifier, then the string input will not be the same as entered. The **`%s` specifier will read only up to the character before the first blank appears**. Thus if someone enters the name "`Sayan Chakraborty`" then only "`Sayan`" will get stored in the array. Therefore **one cannot use `scanf()` along with %s to enter multiple words**.

You can print a string using the printf( ) function and the %s format specifier.

There are **several ways to overcome this problem**. One way is to use another format specifier along with the `scanf()` function and get the required result as shown below:

```
1  /*Program-124: To print a string using scanf() without the %s format specifier*/
2  #include<stdio.h>
3  void main()
4    { char str[80];
5      printf("\nEnter the string: ");
6      scanf("%[^\n]", str);
7      printf("\nThe entered string is %s", str);
8    }
```

Using %[^\n] with scanf( )

You cannot use the scanf( ) function with %s specifier to input a multiword string.

Output

```
Enter the string: Pulak Chand Kundu
The entered string is Pulak Chand Kundu
```

In the above example, the specifier `%[^\n]` indicates that the function will take in all characters, (including white-space characters like blank, tab etc.) until it gets the new-line character '`\n`' i.e. it will input all characters which are not equal to newline i.e. '`\n`'.

The gets( ) function can be used to input a multi-word string

Another form of `scanf()` can be used to restrict input to only lowercase and uppercase alphabets. The function stops taking a character when it gets a character other than the alphabets, as shown:

```
scanf("%[A-Za-z]", str);
```

Thus if someone enters the string "`time2go`", then only the portion `time` will get stored.

Another function, namely the **`gets()` function can be used to input a multiple word string**. The `gets()` function terminates the input when it receives a new-line character '`\n`' (i.e. the Enter key) as input from the user. It then **stores the string by replacing the new-line character** '`\n`' with a null indicator '`\0`' to mark the end of the string. Similar to the `scanf()` function, the argument of `gets()` function is also the array name used to input the string. The syntax for the character array `str[]` is:

Use of gets( ) function

```
gets(str);
```

The following program uses the `gets()` function and **counts the number of words in a sentence.**

To find the number of words

```
1  /*Program-125: To count number of words in a sentence*/
2  #include<stdio.h>
3  void main()
4    { char str[80];
5      int i, count=0;
6      printf("\nEnter the string: ");
7      gets(str);
8      for(i=0; str[i]!= '\0'; i++)
9        if(str[i] == ' ')
10         count++;
11     printf("\nThe number of words in the string is %d", count+1);
12   }
```

The number of words in a sentence is equal to number of blanks separating the words plus 1.

Output

```
Enter the string: Jack of all trades but master of none
The number of words in the string is 8
```

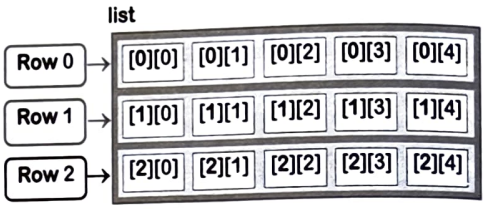
The program-125 inputs a string using the `gets()` function and determines the number of words present in that string. In doing so it counts the number of positions where there is an occurrence of the blank space in the sentence. By counting the number of blanks we can determine the number of words, as a sentence with `n` number of blanks will have (`n+1`) words.

Line-7 uses the `gets()` function to enter the string. The `for` loop of line-8 then reads through the string until it gets the null character '\0'. The `if` statement of line-9 then checks each of the characters given by `str[i]`. If `str[i]` is a blank, i.e. ' ', then it increments the variable `count` by 1. Therefore for the above example, the value of `count` at the end of the string will be 7 as there are 7 blanks in above the sentence.

In line-11, the `printf()` function is used to print the number of words. Note that '1' has been added to `count` to get the number of words from the number of blanks.

14.3 Matrix and Strings

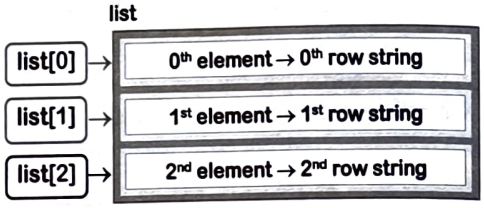
We have seen that we can store a single string in a single array. When we are dealing with a 2D array i.e. a matrix, we can treat each row of a matrix as an array and each such array can be used to store a string. Therefore a **character matrix can be used to store multiple strings**. The first array index of such a matrix defines the row number i.e. the strings number and the second index indicates the maximum characters in each string.



List of strings in matrix representation

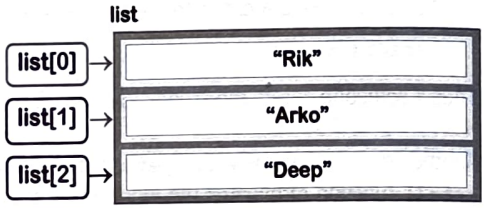
```
char list[3][5] = { {'R', 'i', 'k', '\0'},
                    {'A', 'r', 'k', 'o', '\0'},
                    {'D', 'e', 'e', 'p', '\0'},
                  };
```

3 strings — 5 char each



A matrix of character type can be used to store a list of strings. Each row in the matrix stores a string

The above declaration indicates a character matrix `list[][]`, which can hold 3 strings, with each string containing a maximum of 4 characters (the 5th character is reserved for '\0' to mark the end of each string). Since each word is not exactly 4 characters long, the remaining elements in each row remain unused.



But we have seen earlier that a string can be more conveniently written by writing the string within double quotes (" "). We give below the same string initialisation code shown above but **using double quotes to enclose the strings**.

```
char list[3][5] = {"Rik", "Arko", "Deep"};
```

The above code automatically assigns each string to each row of the `list[]` matrix and inserts the null terminator '\0' at the end of each string. An interesting property of the above initialisation is that **each string can be treated as a single element of a one dimensional array** consisting of 3 elements. Thus each of the 3 strings can be referred to by the row number of the matrix as:

```
list[0] → {"Rik"}
list[1] → {"Arko"}
list[2] → {"Deep"}
```

The following program **reads several strings and calculates the length** of each string.

Calculating lengths of a list of strings

```
1  /*Program-126: To calculate length of multiple strings*/
2  #include<stdio.h>
3  void main()
4    { char names[10][20];
5      int num, i, len;
```

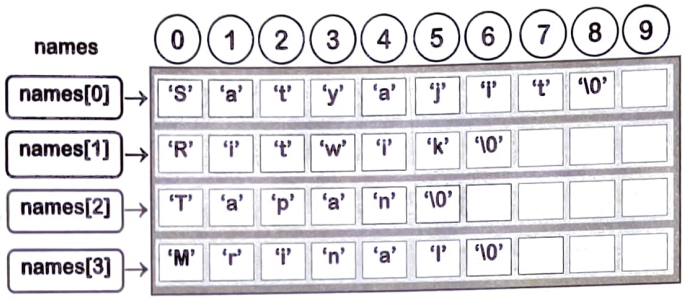
```
    printf("\nEnter the number of words to input (Maximum 10): ");
    scanf("%d", &num);
    for(i=0; i<num; i++)
      {printf("\nEnter name-%d: ", i);
       scanf("%s", names[i]);
      }
    for(i=0; i<num; i++)
      { for( len=0; names[i][len] != '\0'; len++);
        printf"\nLength of %s = %d", names[i], len);
      }
}
```

Output

```
Enter the number of words to input (Maximum 10): 4

Enter name-0: Satyajit
Enter name-1: Ritwik
Enter name-2: Tapan
Enter name-3: Mrinal

Length of Satyajit = 8
Length of Ritwik = 6
Length of Tapan = 5
Length of Mrinal = 6
```



Storing a list of strings

The words to be entered are stored in the matrix `names[10][20]` declared in line-4, which can hold a maximum of 10 strings with each string containing a maximum of 20 characters. Based on this declaration the compiler will reserve `10 x 20 = 200` bytes of memory area to store the strings in the matrix `names[][]`. In line-7, the total number of words (out of a maximum of 10) to be entered is input into the variable `num`.

As discussed earlier, during input of a string in line-10, only the first index of the matrix is specified in the `scanf()` function as `names[i]` to indicate the ith string. This is done as each row of the matrix where each string is stored can be identified by the combination of the matrix name and the row number as `names[i]`.

After the strings are entered, the `for` loop of line-12 is used to read the strings one by one and then count the number of characters in each. For each run of the `for` loop of line-13, the variable `len` is initialised to `0` first. Then using the technique of program-123, the length of the string represented by `names[i]` is calculated and stored in the variable `len`. Line-14 then prints the length of the string `names[i]`. After calculating and displaying a length, the variable `len` is again initialised to `0` in the `for` loop, for calculating the length of the next string.

14.4 Some library functions available to handle Strings

Just like different library functions that are available to handle I/O, mathematical operations, etc. C provides us with **several library functions to handle strings**. String handling includes common tasks like copying two strings [`strcpy()`], concatenating, i.e. joining two strings [`strcat()`], finding the length of a string [`strlen()`], comparing two strings [`strcmp()`], etc. There are other string handling functions but we will restrict ourselves to the most commonly used above four. **Include the header file** `<string.h>` for these.

The `strlen()` function can be used to find the length of a string.

1. `strlen()`: This function is used to **count the number of characters in a string**, excluding the null terminator. It takes only one single string as its argument and returns an integer value indicating the length of the string. The argument can be a variable array or a string constant (like "computer").

The strlen() function

The syntax for the function is: `strlen( string );`

Using the strlen( ) function

```
1  /*Program-127: To find the length of a string using strlen() function*/
2  #include<stdio.h>
3  void main()
4    { char str[10]; int len;
5      printf("\nEnter the word to find its length: ");
6      scanf("%s", str);
7      len = strlen(str);
8      printf("\nThe number of letters in the word = %d", len);
9    }
```

Output

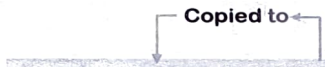
```
Enter the word to find its length: Saptarshi
The number of letters in the word = 9
```

The above program is a **modified version of program-123**, where we had used a `for` loop to calculate the length of a string. The syntax to find the length of a fixed or constant string is to **put the string within double quotes** as shown:

```
len = strlen( "pneumonoultramicroscopicsilicovolcanokoniosis" );
```

The strcpy( ) function

2. `strcpy()` : This function is used to **copy one string into another**. You cannot copy one string into another simply by equating them like other type of variables using the assignment operator. The function copies the contents of the second string i.e. `string2` into the first string i.e. `string1`. The second string can be either a variable array or can be a constant string. The syntax for the function is:




```
strcpy( string1, string2 );
```

The following program is used to **exchange two strings using the** `strcpy()` function.

Exchanging two strings

```
1  /*Program-128: To exchange two strings*/
2  #include<stdio.h>
3  void main()
4    { char str1[80], str2[80], temp[80];
5      printf("\nEnter first string: ");
6      gets(str1);
7      printf("\nEnter second string: ");
8      gets(str2);
9      printf("\nThe entered strings are: %s %s", str1, str2);
10     strcpy(temp, str1);
11     strcpy(str1, str2);
12     strcpy(str2, temp);
13     printf("\nThe swapped strings are: %s %s", str1, str2);
14   }
```

The strcpy( ) function can be used to copy a string type data.

Output

```
Enter first string: Humpty
Enter second string: Dumpty
The entered strings are: Humpty Dumpty
The swapped strings are: Dumpty Humpty
```

The target string is the first argument and the source string is the second argument of the strcpy( ) function.

Three character array type variables called `str1[]`, `str2[]` and `temp[]` are declared in line-4. The first string is entered in line-6 into `str1[]`, and the second string is entered into `str2[]` in line-8 using the `gets()` function.

The strings are printed `str1[]` followed by `str2[]` in line-9. Then from line-10 to line-12 the swapping of the two strings are done. First `str1[]` is copied to the temporary character array `temp[]` using `strcpy()` in line-10. Then `str2[]` is copied to `str1[]` using `strcpy()` in line-11. Finally the string stored in `temp[]` is copied to `str2[]` using `strcpy()` in line-12. The swapped strings are displayed in line-13.

3. `strcmp()`: This function is **used to compare two strings**. By comparing two strings is to see if two strings are identical or **if they are in proper alphabetical order or not**. By proper alphabetical order for example we mean if a string starting with '`b`' (like book) is after a string starting with '`a`' (like apple) or not. Thus {**a**pple, **b**ook} are in order but {**b**ook, **a**pple} are not in alphabetical order. The syntax for comparing two strings `string1` and `string2` using `strcmp()` is:

String Compare function strcmp( )

```
strcmp(string1, string2);
```

Depending upon the above possibilities, the `strcmp()` function works in the following ways:

Working of strcmp( )

a) **Both the strings are identical**: The function **returns 0** (and not 1, remember this carefully as the inverse of this value can be used in the condition of a `if` statement to test two strings to be same).

b) `string1` **is in alphabetical order with** `string2`: The function returns a **negative number**.

c) `string1` **is not in alphabetical order with** `string2`: The function returns a **positive number**.

Thus by checking for the sign of the returned value you can determine whether `string1` is equal, higher or lower than `string2` in alphabetic order.

The strcmp( ) function can be used to check if one string is equal, higher in alphabetic order, or lower in alphabetic order than another string.

The logic of the `strcmp()` function may at first seem confusing. But the way of working of the function will make things clear. The `strcmp()` function in reality finds the difference in the ASCII values of the first non-matching characters in the two strings. Thus if `string1` (say Sujoy) is in alphabetical order with `string2` (say Sumon), then the first non-matching character '`j`' (Sujoy) with ASCII value 106 in `string1` minus the first non-matching character '`m`' (Suman) with ASCII value 109 in `string2` results in a negative number (`106-109 = -3`).

We now give below a program to **sort a list of strings in alphabetical order**. We will be using the `strcmp()` function to compare two strings and then the `strcpy()` function to interchange pairs of strings in case they are not in alphabetic order.

Sorting a list of strings

```
/*Program-129: Sorting a list of strings*/
#include<stdio.h>
#include<string.h>
void ascending(char names[][20], int n);
void main()
  { int num, i;
    char names[100][20];
    printf("\nEnter number of names in the list: ");
    scanf("%d", &num);
    for(i=0; i<num; i++)
       {printf("\nEnter name-%d: ", i);
        fflush(stdin);
        gets(names[i]);
       }
    ascending(names, num);
    printf("\nList of names in ascending order:");
    for(i=0; i<num; i++)
        printf("\n%s", names[i]);
  }
```

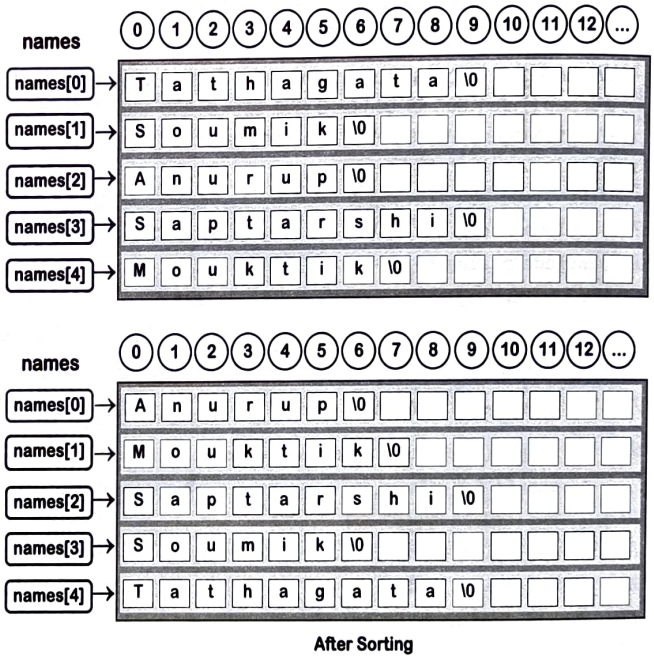
The strcmp( ) function returns 0 for identical strings, a +ve number for strings not in order, and a –ve number for strings in alphabetic order.

```
20 void ascending(char list[][20], int n)
21   { char temp[20];
22     int i, j;
23     for(i=0; i<n-1; i++)
24       for(j=i+1; j<n; j++)
25          { if( strcmp(list[i], list[j]) >0 )
26                {strcpy(temp, list[i]);
27                 strcpy(list[i], list[j]);
28                 strcpy(list[j], temp);
29                }
30          }
31   }
```

Output

```
Enter number of names in the list: 5
Enter name-0: Tathagata
Enter name-1: Soumik
Enter name-2: Anurup
Enter name-3: Saptarshi
Enter name-4: Mouktik
List of names in ascending order:
Anurup
Mouktik
Saptarshi
Soumik
Tathagata
```

In line-7 we have declared a character matrix called `names[100][20]` to store a maximum of 100 names with each name up to 20 characters long. In line-9 the actual number of names are input into the variable `num`. The `for` loop of line-10 is then used to enter the names in the list. The diagram on the right shows the state of the matrix after inputting the names. The `fflush(stdin)` function in line-12 is used to clear the input buffer before entering the names in line-13 using the `gets()` function.



After Sorting

In line-15, the `ascending()` function is called to sort the list of names in ascending order. The matrix `names` and the number of names stored i.e. `num` are passed to the function. The **formal argument** `list` receives the matrix in the function header. Note how the variable `list` which receives the `names` matrix is declared. You need to use the two pairs of square braces to indicate the variable received as a matrix. Moreover you need to specify the second index in `list` as `list[][20]`. **This is a MUST**. Though you may or may not mention the first index.

Within the function, the `for` loops of lines-23 and 24 are used to sort the list of names in ascending order using **Selection Sort** method.

In line-25, the `strcmp()` function is used to compare the string `list[i]` with `list[j]`. As we have discussed earlier, if `list[i]` i.e. the ith name in the list is not in alphabetical order with respect to `list[j]` i.e. the jth name in the list, then the function returns a positive value and the `if` condition becomes true.

When the `if` condition is true, the strings `list[i]` and `list[j]` are exchanged as per the logic of the program 128 using the `strcpy()` function in lines-26 to 28.

Finally the sorted list is printed using the `for` loop of line-17.

- `strcat()` : This function is used **to join two strings** i.e. one string gets appended or joined at the end of the other string. The syntax for the function is:

```
strcat(string1, string2);
```

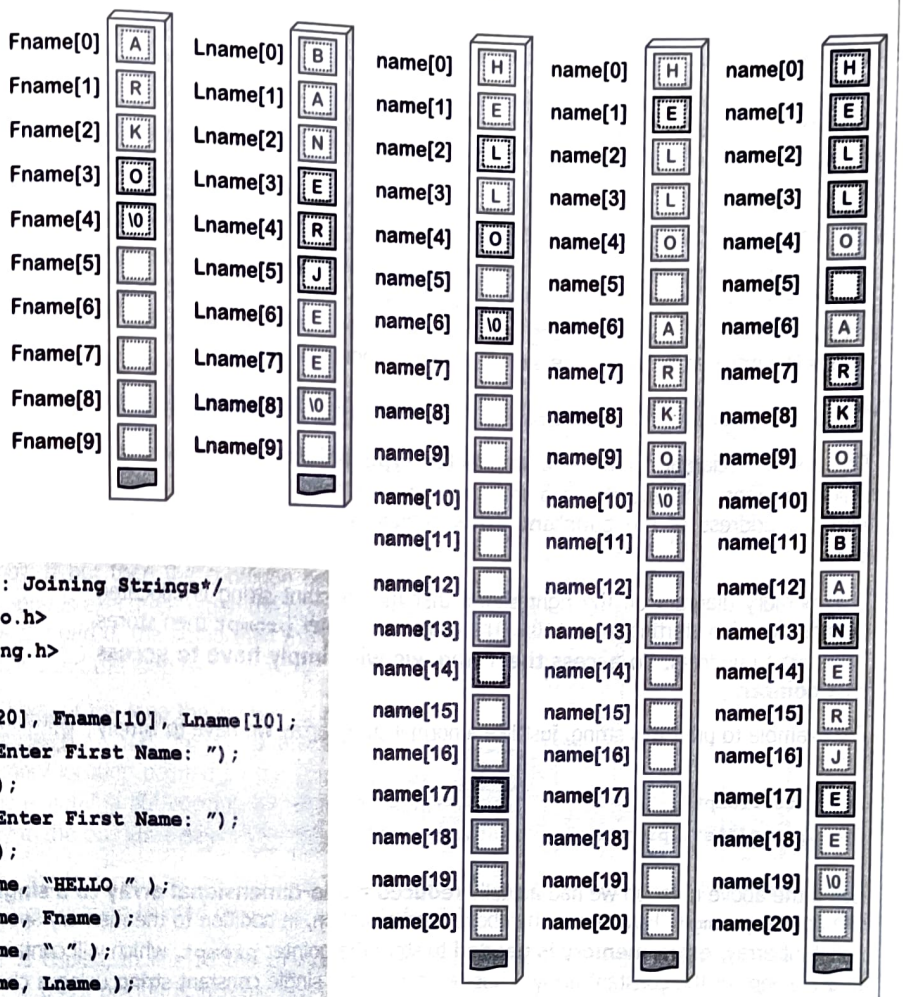
Both `string1` and `string2` are character arrays and the second string i.e. `string2` gets added after `string1`. Hence care should be taken so that `string1` **is declared with sufficient size to contain both** `string1` **and** `string2`. The `strcat()` function works by replacing the null indicator '\0' at the end of the `string1` with the starting character of the second string.

String Joining function strcat( )

A strcat( ) function is used to join two strings.

The second string argument gets appended at the end of the first string argument in strcat( ).

The following example **joins three strings** namely `name`, `Fname`, and `Lname`. The constant string "HELLO " is first copied to `name`. The remaining strings i.e. `Fname` and `Lname` are getting added to this constant string to form a final greeting.




```
1  /*Program-130: Joining Strings*/
2  #include<stdio.h>
3  #include<string.h>
4  void main()
5  { char name[20], Fname[10], Lname[10];
6    printf("\nEnter First Name: ");
7    gets(Lname);
8    printf("\nEnter First Name: ");
9    gets(Lname);
10   strcat( name, "HELLO " );
11   strcat( name, Fname );
12   strcat( name, " " );
13   strcat( name, Lname );
14   printf("\n%s", name);
15 }
```

Output

```
Enter First Name: ARKO
Enter Last Name: BANERJEE
HELLO ARKO BANERJEE
```

3 different character arrays namely `name[26]`, `Fname[20]`, and `Lname[20]` have been declared in line-5.

In lines-6 and 7 the two strings are entered into the variables `Fname` and `Lname` using the `gets()` function.

Use double quotes to enclose a single blank character when using a blank space as an argument of the strcat( ) function.

In line-8, the constant string "`HELLO `" is copied to the blank string array `name[]` using `strcat()`. In line-9, the string stored in `Fname` is then joined to the existing string in `name[]`. In line-10, a blank character is added after the first name to separate the first name from the last name. Finally in line-11, the string stored in `Lname` is added to the existing string in `name[]`. Line-12 then prints the composite string stored in `name[]`.

The blank is added to separate the two names in the final string. **Note the double quotes used** to enclose the **single blank space**. Since the `strcat()` function joins strings, we cannot use a single quote to indicate a blank, but a pair of double quotes to indicate a **blank string**. The memory diagram in the previous page gives the memory content of the arrays `Fname[]` and `Lname[]`. The three diagrams for the `name[]` array show how the three individual strings are added one after the other to `name[]` to get the final string.

Also keep in mind that the string `name[]` **should be made sufficiently large** to accommodate the constituent strings along with blank characters and the null character.

14.5 Pointer and Strings

Relation between Pointer and strings

A string is basically a **character array** and hence it is natural that we can **control strings efficiently using pointers** as array operations can be done efficiently using pointers. A constant string can be declared using the following string array declaration:

```
char prompt[] = "Press Enter"; /*Array initialised by string */
```

The above character array will allocate 12 bytes of memory (1 byte for each character) including one byte for `'\0'` to store the above string. However instead of using a character array, we can also **use a character pointer to store the same string** as shown below:

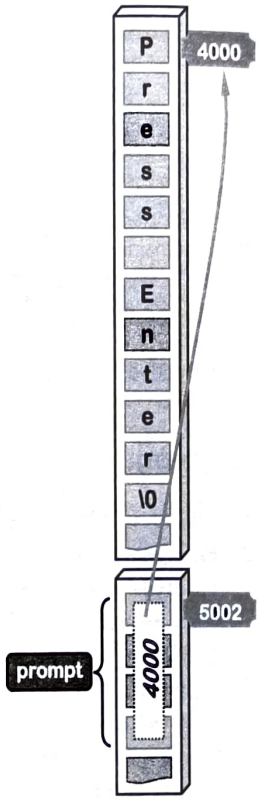
Using pointer to declare a constant string

```
char *prompt = "Press Enter"; /*Pointer initialised by string*/
```

The above declaration creates a **character type pointer called** `prompt`, which can store the address of a character type variable. It is then initialised with the address of the **constant array** containing the characters "`Press Enter`".

A constant string can be stored using a character pointer.

The memory diagram on the right shows that the constant string is allocated memory location starting from `4000`. The character pointer `prompt` then stores this starting address. **To access the string, we will simply have to access this pointer**.



For example to print this string, just like a normal string array we have to simply type:

```
   puts(prompt);          /*Prints the string stored in prompt*/
OR printf("%s", prompt);  /*Prints the string stored in prompt*/
```

Using the above notation we had actually **reduced a one-dimensional array to a single variable** namely the pointer `prompt`. However in the above representation, in addition to the memory space used to store the constant array, **extra memory is needed** to store the pointer `prompt`, which will contain the address of the first element of the constant array. Thus representing a single constant string using a character pointer may not be of much help. The major advantage of this approach comes when we try to store a set of constant strings (to be covered in book-2).

As pointers provide an efficient and compact technique to access arrays, **programming operations with character strings are almost invariably done with pointers**. The following program simply copies the contents of one string called `InputStr` into another string called `OutputStr`.

String Copy using pointers

```
1  /*Program-131: String Copy using Pointers*/
2  #include<stdio.h>
3  void main()
4  { char InputStr[80], OutputStr[80];
5    char *p_in = InputStr, *p_out = OutputStr;
6    printf("\nEnter the string to copy: ");
7    gets(InputStr);
8    while( *p_in != '\0')    /*Loop runs till value pointed by p_in is not equal to null*/
9      (*p_out = *p_in;       /*Content pointed by p_in copied to memory pointed by p_out*/
10      p_out++;              /*p_out incremented to point to the next character*/
11      p_in++;               /*p_in incremented to point to the next character*/
12     )
13   *p_out = '\0';
14   printf("The copied string is: %s", OutputStr);
15 }
```

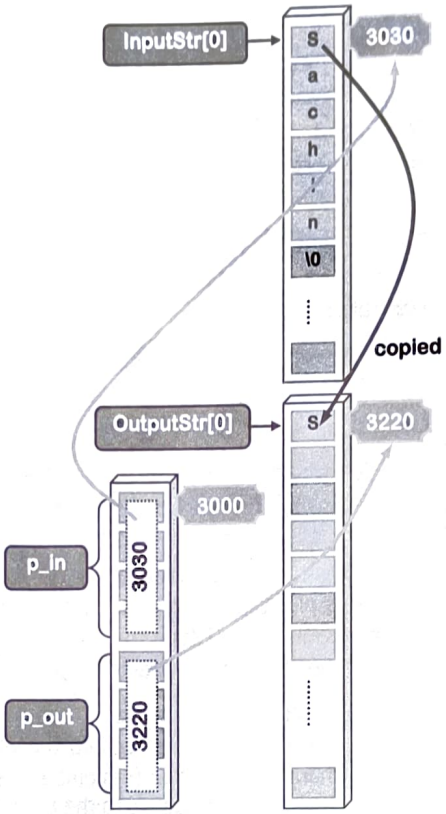
Character pointers initialised with the address of the 0th elements of the character arrays

Output of the above program:

```
Enter the string to copy: Sachin
The copied string is: Sachin
```

Working of String Copy

In line-4 we have declared two string type variables `InputStr` and `OutputStr`. In line-5 we have declared two **character type pointers** `p_in` and `p_out`. Along with the declaration, `p_in` is assigned the address of the character array `InputStr` and `p_out` is assigned the address of the character array `OutputStr`. Hence `p_in` stores the starting address of `InputStr` and `p_out` contains the starting address of `OutputStr` as shown on the right.



The input string is entered within the array `InputStr` in line-7 using a `gets()` statement. In our output example we have entered the string "`Sachin`" within `InputStr` as shown in the diagram on the right.

The **main part** of the above program **is the `while` loop** in line-8 which is **engaged in copying the string**. The `while` loop condition checks if the value pointed by pointer `p_in` (i.e. `*p_in`) is '`\0`' or not. If not then the condition becomes true and the body of the loop is executed. When the pointer reaches the '`\0`' character at the end of the string then the `while` loop condition becomes false and the loop terminates.

In line-9 within the body of the loop the content of the location pointed by the pointer `p_in`, i.e. `*p_in` is assigned to the contents of the memory location pointed by the pointer `p_out` i.e. to `*p_out`. This is nothing but copying the character from the `InputStr` array to the `OutputStr` array.

In our example when the loop runs for the first time the character 'S' at `InputStr[0]` is pointed by `p_in`. It is copied to the memory location pointed by `p_out` i.e. to `OutputStr[0]`.

After one character is copied, in line-10 the `p_out` pointer is incremented to point to the next character position. In line-11 the `p_in` pointer is incremented to point to the next character position. The loop condition is again checked. If the condition is true, the next character is copied and the pointers incremented.

When the '`\0`' character is reached by the pointer `p_in` the loop terminates. In line-13 the '`\0`' character is inserted at the array location pointed by `p_out` to mark the end of the copied string.

The entire loop can be written in a **compact manner** as shown below. Try to analyse the code on your own!

```
8  while(*p_out++ = *p_in++);        /*Value assigned first, then pointer incremented*/
```

14.6 Some worked out problems

The next program **checks the number of characters entered** and terminates the string if the number of characters crosses the maximum length of the string array including the '\0' character.



Counting characters during string input

```
1   /*Program-132: String input up to specific number of characters only*/
2   #include<stdio.h>
3   #define MAX 10
4   int main()
5   { char str[MAX], letter;
6     int count=0;
7     printf("\nEnter the string: ");
8     while( 1 )
9       {letter = getchar();
10       if (letter=='\n')
11           break;
12       if (count==MAX-1)
13           { printf("\nString too large and terminated...");
14             break;
15           }
16       str[count] = letter;
17       count++;
18      }
19    str[count] = '\0';
20    printf("The input string is: %s", str);
21    return 0;
22  }
```

Output1:

```
Enter the string: beautiful
The input string is: beautiful
```

Output2:

```
Enter the string: entertainment
String too large and terminated...
The input string is: entertain
```

In line-3 we have defined **MAX** as 10. Hence the program will input strings up to a maximum length of 9 characters plus one null character. The string array **str[]** declared in line-5 will store the input string. The variable **count** is initialised to 0 in line-6 and is used to serve as the index for the string. It also serves as a count of the number of characters input.

The **while** loop of line-8 is used to enter the string, character by character. The **getchar()** function of line-9 is used to enter a character into the **char** variable **letter**. Line-10 checks if this input character is a newline character i.e. Enter (remember that the Enter key is pressed when a string entry in completed and the '**\n**' character corresponds to the Enter key). If so, the **break** statement of line-11 terminates the loop.

The **if** statement of line-12 then checks if the present **count** value is equal to the last index in the array. If so, as no more characters can be input in the array, it prints the error message and breaks out of the loop in line-14. Remember that for a proper string you can input a maximum of characters one less than the size of the array as the position at the last index is used for storing the '\0' character.

Otherwise the input character is stored in the array at position **count** in line-16. In line-17 the **count** variable is incremented by 1 to point to the next array index.

When the loop terminates by any one of the **break** statements, in line-19 a '\0' character is forcibly stored at position **count** (remember that when the loop terminates, **count** points to the index after the last valid character stored in the string array).

The next program finds the **number of occurrences of a particular character in a string**. This is similar to the program to count the number of blanks in a string.



Count number of occurrences of the alphabet 'e'

```
/*Program-133: To count number of occurrences of the alphabet 'e' in a string*/
#include<stdio.h>
void main()
  { char str[80];
    int i, count=0;
    printf("\nEnter the string: ");
    gets(str);
    for(i=0; str[i]!= '\0'; i++)
       {if(str[i] == 'e' || str[i] == 'E')
           count++;
       }
    printf("\nThe number of 'e' in the string is %d", count);
  }
```

Output

```
Enter the string: Evening walk is good for health
The number of 'e' in the string is 3
```

In the above program, the `if` statement of line-9 checks the character `str[i]` to be the character `'e'` or `'E'` by using the logical OR operator. Thus for the input string in the above example, it finds a `'E'` in the position `str[0]` and two `'e'` in the positions `str[2]` and `str[26]` and prints accordingly in line-12.

The **next program is a modified form of the above program** where the character to find is input by the user instead of a fixed character written inside the program.



Count number of occurrences of any alphabet

```
/*Program-134: To count number of occurrences of any alphabet in a string*/
#include<stdio.h>
void main()
  { char str[80], letter;
    int i, count=0;
    printf("\nEnter the string: ");
    gets(str);
    printf("\nEnter alphabet to find: ");
    fflush(stdin);
    letter = getchar();
    if (letter>=97 && letter<=122)
       letter = letter-32;
    for(i=0; str[i]!= '\0'; i++)
      { if( (str[i] == letter) || (str[i] == (letter+32)) )
           count++;
      }
    printf("\nThe number of '%c' in the string is %d", letter, count);
  }
```

Output

```
Enter the string: She sells sea shells on the sea shore
Enter alphabet to find: s
The number of 's' in the string is 8
```

In the above program the character to search is entered into the variable `letter` in line-10 using the `getchar()` function (Note the use of the function `fflush(stdin)` before using `getchar()`). As the character entered can either be in lower or in upper case, it is first converted to upper case for uniformity by the `if` statement of line-11 and 12.

Then the `for` loop of line-13 is used to check the occurrence of the character indicated by `letter`. Note that

both the small and capital alphabets are checked by checking the variable `letter` (for capital alphabet) and `letter+32` (for the small alphabet) using the logical OR operator.

The next program **counts the number of words** in a sentence **separated by any number of blanks.**



Counting words separated by multiple blanks

```
1  /*Program-135: To count number of words in a sentence with multiple blanks*/
2  #include<stdio.h>
3  void main()
4     { char str[80];
5       int i, count=0;
6       printf("\nEnter the sentence: ");
7       gets(str);
8       for(i=0; str[i]!= '\0'; i++)
9          {if(str[i] == ' ' && str[i+1] != ' ')
10             count++;
11         }
12      printf("\nThe number of words in the sentence is %d", count+1);
13    }
```

Output

```
Enter the sentence: Happiness   in  intelligent    people  is a   rare    trait
The number of words in the sentence is 8
```

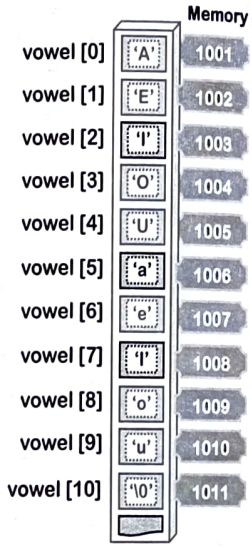
The string to search is entered in line-7. The `for` loop of line-8 scans the characters to count the words. For a set of characters to be a word it must be preceded by a blank space. This condition is checked by the `if` statement in line-9. So if `str[i]` is a blank and `str[i+1]` is not a blank, then the counter `count` is incremented. The condition will be false for any two consecutive blank spaces.

The next program is used to **count the number of vowels in a sentence.**



Counting number of vowels in a string

```
1  /*Program-136: To count the number of vowels in a string*/
2  #include<stdio.h>
3  void main()
4     { char str[80], vowel[11] = "AEIOUaeiou";
5       int i, j, count=0;
6       printf("\nEnter the string: ");
7       gets(str);
8       for(i=0; str[i]!= '\0'; i++)
9          {for(j=0; vowel[j]!='\0'; j++)
10            if(str[i] == vowel[j])
11               {count++;
12                break;
13               }
14         }
15      printf("\nThe number of vowels in the string is %d", count);
16    }
```



Output

```
Enter the string: Beauty lies in the eyes of the beholder
The number of vowels in the string is 14
```

In the above program we have declared two strings in line-4. The first one to input the string `str[]`, and another one called `vowel[]` to store all the vowels. The string `vowel[]` has been initialised with a constant string "`AEIOUaeiou`" that contains all vowels both in upper and lowercase. The '`\0`' is automatically inserted at the end of the string as shown above.

The string to check is input in line-7. The `for` loop of line-8 then scans the string and takes a character `str[i]` for every `i` till it gets the '`\0`' character. The `for` loop of line-9 is within the outer `for` loop. It is used to scan the array `vowel[]` with loop index `j`.

The `if` statement of line-10 is part of this second `for` loop. It is used to compare a particular `str[i]` with all `vowel[j]` by changing the array index `j` within the second `for` loop. In case it finds a match, i.e. a particular `str[i]` matches with one of the `vowel[j]`, the counter `count` is incremented by 1 and the control breaks out of the inner loop. The program then increments `i` and repeats the process for the next `str[i]`. Line-15 finally prints the number of vowels in the input string.

Instead of using the constant string `vowel`, you can use an `if` statement like the one shown below to compare the value in `str[i]` with any of the vowels. But the code requires a **lengthy condition** checking.

```
if(str[i]=='A' || str[i]=='E' || str[i]=='I' || str[i]=='O' || str[i]=='U' || str[i]=='a'
   || str[i]=='e' || str[i]=='i' || str[i]=='o' || str[i]=='u')
```

The next program is used to **check if a string is a palindrome** or not.



Program to check for a palindrome

A **palindrome** is a string that reads the same from both ends. Example 'liril', 'madam', etc.

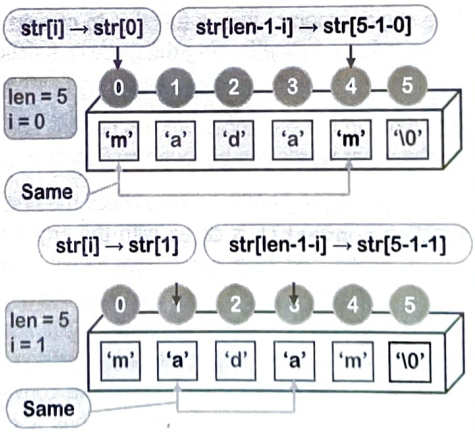
```
1  /*Program-137: To check for a palindrome*/
2  #include<stdio.h>
3  void main()
4    { char str[80];
5      int i, len, flag=0;
6      printf("\nEnter the string: ");
7      gets(str);
8      for(len=0; str[len]!= '\0'; len++);
9      for(i=0; i < len/2; i++)
10         { if(str[i] != str[len-i-1])
11             {flag=1;
12              break;
13             }
14         }
15     if(flag==0)
16         printf("\nThe string is a palindrome");
17     else
18         printf("\nThe string is not a palindrome ");
19   }
```

Output

```
Enter the string: MADAM
The string is a palindrome
```

A palindrome is a word that reads the same from both the sides. For example the words "madam", "liril", "dad", "eye", "racecar", "malayalam" etc. are all palindromes. The above program enters the string to check for a palindrome in line-7. The length of the string is then found out using the method discussed in program-123 using the `for` loop of line-8. The variable `len` is used to store the length of the string. The `for` loop of line-9 is then used to check for the palindrome.

The example on the right gives an idea of the logic of the program. To check for a palindrome, it is sufficient to check for the letters of the word from either end in pairs till you reach the centre of the word. If the word is a palindrome, the letter pairs from either ends will match. For example for "madam", the pair of 'm' and the pair of 'a' from either end are identical. After that we reach the centre character 'd' and need not check any further. However for the word "modem", though the outer character pair 'm' matches, the next outer characters 'o' and 'e' are different. Hence "modem" is not a palindrome.



For the word "madam" with `len=5`, to get the character pairs from either end in `str[]` we use the index pairs as:

`str[0]` and `str[4]` i.e. for `i=0`, compare `str[0]` and `str[len-1-i]` i.e. `str[5-1-0]` → `str[4]`

`str[1]` and `str[3]` i.e. for `i=1`, compare `str[1]` and `str[len-1-i]` i.e. `str[5-1-1]` → `str[3]`

Note the condition part of the `for` loop. It is taken as `i<len/2` as it is sufficient to check up to the centre of the string. Thus for the word "madam", with `len=5`, the condition will be `i<(5/2)` i.e. `i<2` (considering integer division). Thus the index `i` will vary from 0 to 1 and check the characters `str[0]` and `str[1]` along with their respective pairs at the other end.

If the string is a palindrome, then the `if` condition of line-10 will never be true and the value of the variable `flag` remains at 0. However, if the word is not a palindrome, then the condition will be true the moment it encounters two letter pairs that are different and will change the value of the variable `flag` to 1. It will then `break` out of the loop as there is no need to check any further.

The `if` statement of line-15 then checks the value in `flag` and prints whether the word is a palindrome or not. If `flag=0`, then the word is a palindrome, else if `flag=1`, it is not a palindrome.

The next program is used to **convert the first letter of every word to uppercase.**



Capitalize first character of each word

```
1  /*Program-138: To convert the first character of every word to capital*/
2  #include<stdio.h>
3  char convert(char letter);
4  void main()
5    { char str[80];
6      int i;
7      printf("\nEnter the string: ");
8      gets(str);
9      str[0]=convert(str[0]);
10     for(i=0; str[i]!='\0'; i++)
11          { if(str[i] == ' ')
12               str[i+1]=convert(str[i+1]);
13          }
14     printf("\nThe converted string is: %s", str);
15   }
16 char convert(char letter)
17   { if(letter>=97 && letter<=122)
18        letter = letter-32;
19     return letter;
20   }
```

Output

```
Enter the string: arise awake and stop not till your goal is reached
The converted string is: Arise Awake And Stop Not Till Your Goal Is Reached
```

The above program uses a function to convert the first character of every word in a sentence to capital. The string is input in line-8 using the `gets()` function.

We know that in a sentence a new word starts after a blank space. This logic is true for all words, except the first word in the sentence, which is not preceded by blank. Thus except the first word the starting character of all the other words can be found by finding the blank and checking for the character after the blank.

Line-9 of the program is used to check the first character of the sentence by checking `str[0]`. To do so, the function `convert()` is called with the argument as `str[0]`. In our above example, for the entered string "arise awake and stop not till your goal is reached", `str[0]='a'`. The variable `letter` in the function `convert()` is used to accept the value in `str[0]`. Thus `letter` = 'a' for the first character of the sentence. The `if` statement in line-17 checks if the character stored in the variable `letter` is in lowercase. In this example as 'a' is in lowercase the condition is true and `32` is subtracted from `letter` to convert it to uppercase 'A'. Line-19 then returns the converted character stored in `letter`. The returned character is reassigned back to `str[0]` in line-9. Thus 'A' gets stored in `str[0]` now.

The `for` loop of line-10 is then used to scan the string to search for a blank. The `if` statement of line-11 then checks if `str[i]=' '`. If the condition is true then the next character should be the starting letter of

the next word. Thus, if the condition is true the next character i.e. `str[i+1]` is sent to the function `convert()`. The function `convert()` then checks if the character is in lowercase. If so, it converts it to uppercase and sends it back to `main()`. Within the `main()` function the capital letter is assigned back to `str[i+1]`. Line-14 then prints the converted sentence.

The **next program is used to find the occurrence of a given word in a given sentence**. To check for a whole word, there can be three different cases. One, when the word is located as the first word of the sentence. Second, when the word is located in between; and thirdly when the word is located at the end of the sentence as the last word.



To find the number of "the" in a sentence

```
1  /*Program-139: To find the occurrence of the word 'the' in a sentence*/
2  #include<stdio.h>
3  void main()
4    { char str[80];
5      int i, count=0;
6      printf("\nEnter string: ");
7      gets(str);
8      for(i=0; str[i]!='\0'; i++)
9        { if(str[i]>=97 && str[i]<=122)
10           str[i]=str[i]-32;
11       }
12     if( (str[0]=='T') && (str[1]=='H') && (str[2]=='E') && (str[3]==' ') )
13        count++;
14     for(i=0; str[i]!='\0'; i++)
15        {if(str[i]==' ')
16           {if((str[i+1]=='T') && (str[i+2]=='H') && (str[i+3]=='E') && (str[i+4]==' '))
17               count++;
18           }
19        }
20     if( (str[i-1]=='E') && (str[i-2]=='H') && (str[i-3]=='T') && (str[i-4]==' ') )
21        count++;
22     printf("\nTotal number of the word 'the' in the sentence is %d", count);
23   }
```

Output

```
Enter string: The is an article and there are no sentences that end with the word the
Total number of the word 'the' in the sentence is 3
```

Program-138 is written to find how many times the word "the" has occurred in a sentence. The string is entered in line-7 using `gets()`. The `for` loop of line-8 is used to **convert the sentence to uppercase** to simplify the code. This is done to simplify the search, as there can be various combinations of lower and uppercase in a string and we had to search for all those combinations. After the conversion, to find the word "the", there can be three different situations as discussed earlier:



Different situations for occurrence of "THE"

- "THE" at the beginning (blank after the word "THE")
- "THE" in between (blank on either side of word "THE")
- "THE" at the end (blank before the word "THE")

The example sentence typed above gives us all the three conditions. In line-12, the `if` statement checks for the presence of the word **"the" at the beginning** of the sentence. To do so it checks if `str[0]='T'`, `str[1]='H'`, `str[2]='E'`, and finally `str[3]=' '`, as the first word ends with a blank. This differentiates it from the word THERE. If the condition is satisfied, it increments `count` by 1.

The `for` loop of line-14 then checks for the presence of the word **"the" in between**. In doing so, the `if` statement of line-15 looks for the presence of the blank character, as a new word starts after the blank. If it finds a blank at position `i`, then it looks for the next 4 characters to be 'T', 'H', 'E', and ' ' using the `if`

statement of line-16. The conditions are joined by logical AND operations within the `if` statement. With each occurrence of the word "THE", the variable `count` is incremented by 1.

Finally the `if` statement of line-20 is used to check for the occurrence of the word **"the" at the end**. For the end of the sentence, the character after the word is '\0' and the character preceding the word "the" is a blank. When the `for` loop of line-14 ends, the loop counter `i` points to the null character '\0'. We can take help of this and use the counter `i` to check the last word. If the last word is "THE", then the letter `str[i-1]='E'`, `str[i-2]='H'`, `str[i-3]='T'`, and `str[i-4]=' '` and `count` gets incremented by 1. Finally the number of occurrences is printed in line-22.

The **next program** is used to **find the presence of a given sequence of letters in a string**. Both the string and the sequence are taken as user inputs. The program counts the number of times the given sequence occurs in the string.



Checking a given sequence of characters

```
1  /*Program-140: To check for a given sequence of letters in a string*/
2  #include<stdio.h>
3  #include<string.h>
4  void upper(char string[]);
5  int main()
6    { char str[80], secq[20];
7      int i, j, k, flag, len1, len2, count=0;
8      printf("\nEnter the string: "); gets(str);
9      printf("\nEnter the sequence: "); gets(secq);
10     upper(str);
11     upper(secq);
12     len1 = strlen(str);
13     len2 = strlen(secq);
14     for(i=0; i<(len1-len2+1); i++)
15          { if(str[i]==secq[0])
16               {flag=1;
17                for(j=0, k=i; secq[j]!='\0'; j++, k++)
18                   {if(str[k] != secq[j])
19                      {flag=0; break;}
20                   }
21                if(flag==1)
22                   count++;
23               }
24          }
25     printf("\nTotal number of sequences '%s' in the string is %d", secq, count);
26     return 0;
27   }
28 void upper(char string[])
29   { int i;
30     for(i=0; string[i]!='\0'; i++)
31        { if(string[i]>=97 && string[i]<=122)
32             string[i]=string[i]-32;
33        }
34   }
```

Output

```
Enter string: Trees are there to restore the oxygen in the atmosphere
Enter the sequence: re
Total number of sequences 'RE' in the string is 6
```

The string and the sequence are input in line-8 and 9 respectively. In line-10 and 11 the strings are converted to uppercase using the user defined function `upper()` of line-28. In line-12 and 13 the length of the strings `str` and `secq` are calculated and stored in `len1` and `len2` respectively. The `for` loop of line-14 checks for

the presence of the sequence in the string. The loop runs up to the difference of the lengths of the two strings as there is no point in checking for the presence of the sequence when the number of characters left to check in `str` is less than the length of the sequence.

Line-15 checks if the starting character of the sequence matches with `str[i]` for a given index `i`. If so, a variable `flag` is initialised to 1 in line-16 and the `for` loop of line-17 is used to compare each character of the sequence with each character of the string `str` starting from the current string position given by `i`. If a mismatch occurs, the `if` condition of line-18 becomes false. In that case the variable `flag` is changed to 0 and the inner `for` loop is exit using the `break` statement of line-19. In line-21 the value of `flag` is checked. A '0' value of `flag` indicates that the `if` statement of line-18 had become true i.e. there was a mismatch in sequence. A '1' value of the `flag` indicates that there was no mismatch and a complete sequence was found. In that case the variable `count` is incremented in line-22. The final count is printed in line-25.

The **next program** is used to **find the presence of a given word entered by the user in a string.**



To find a given word in a given string

```
1  /*Program-141: To find the presence of a given word in a string*/
2  #include<stdio.h>
3  #include<string.h>
4  void upper(char string[]);
5  int main()
6    { char str[80], word[20];
7      int i, j, k, count=0, len1, len2, flag;
8      printf("\nEnter the string: "); gets(str);
9      printf("\nEnter the word: "); gets(word);
10     upper(str);
11     upper(word);
12     len1 = strlen(str);
13     len2 = strlen(secq);
14     for(i=0; i<(len1-len2+1); i++)
15         { if(str[i]==word[0])
16             {flag=1;
17              for(j=0, k=i; word[j]!='\0'; j++, k++)
18                 {if(str[k] != word[j])
19                     {flag=0; break;}
20                 }
21              if(flag==1)
22                 {if ( (i==0 || str[i-1]==' ') && (str[k]==' ' || str[k]=='\0') )
23                      count++;
24                 }
25             }
26         }
27     printf("\nTotal number of words '%s' in the sentence is %d", word, count);
28     return 0;
29   }
30 void upper(char string[])
31   { int i;
32     for(i=0; string[i]!='\0'; i++)
33       { if(string[i]>=97 && string[i]<=122)
34           string[i]=string[i]-32;
35       }
36   }
```

Output

```
Enter string: The other side of the moon is there for us to imagine
Enter the word: the
Total number of words 'THE' in the sentence is 2
```

Program-141 is exactly same as program-140 with the only difference being in lines 21 to 24 which are reproduced in the next page for reference:

```
21              if(flag==1)
22                  (if ( (i==0 || str[i-1]==' ') && (str[k]==' ' || str[k]=='\0') )
23                      count++;
24                  )
```

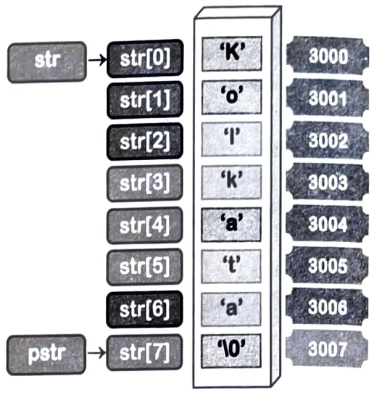
As we have to check for a word instead of a sequence, after identifying the sequence in the string we have to see if it represents a full word or is part of a word. That is, after we find a match in the sequence of characters in the word we need to see if the sequence is forming a word. To do that if **flag** is equal to 1 (sequence match found) the **if** condition of line-24 checks several conditions. For the sequence to be a word it can be the first word in the string or it must be preceded by a blank character. This condition is checked by the part of the condition **(i==0 || str[i-1]==' ')**. **i==0** checks for the first word and **str[i-1]** checks for the preceding character. Similarly for the sequence to be a word, it should be followed by a blank character or by the '\0' character. These conditions are checked by the second part of the overall condition i.e. by **(str[k]==' ' || str[k]=='\0')**. The index **k** is used as it points to the location in the string **str** where the sequence for the word ends. Both these parts should be true simultaneously for the sequence to be a word, and hence are joined by the AND operator **&&**.

The following example calculates the **length of a string in a character array** using pointers.

String length using pointers

```
1   /*Program-142: String Length using Pointers*/
2   #include<stdio.h>
3   void main()
4     { char str[]="Kolkata", *pstr;
5       int length;
6       pstr = str;
7       while(*pstr != '\0')
8           pstr++;
9       length = pstr - str;  /*Difference of pointers*/
10      printf("\nString Length = %d", length);
11    }
```

Starting address of character array i.e. str[0] or str assigned to char pointer pstr



Output:

```
String Length = 7
```

In line-4 we have defined a string array **str[]** and initialised it with the fixed string "**Kolkata**". Let us assume the string starts from the memory location **3000** and ends at **3007** (including the **NULL**). We have also declared a character pointer **pstr**, which can point to the address of a character type variable. In line-5 we have defined an integer variable **length** to store the length of the string.

In line-6, we have assigned the starting address of the character array **str[]** to the character pointer **pstr** (remember, name of an array indicates the address of its 0th element). In this example, therefore **pstr** will point to the address location **3000**.

In line-7, the **while** loop is used to find the position of the **NULL** character in the string array. Within the condition part of the **while** loop, the content of the memory location pointed to by the pointer **pstr** (i.e. ***pstr**) is checked for a null character i.e. '\0'. The loop will run as long as it is not a null. The loop stops when ***pstr** is '\0'. For the first 7 positions of the array i.e. from **str[0]** to **str[6]**, the content is something other than '\0'. Hence the loop condition remains true. The code **pstr++** in line-8 is executed each time the condition is found true. Starting from the value **3000**, with each **pstr++** the value in the pointer **pstr** will increase by 1 byte (as a character occupies 1 byte and **pstr++** means **pstr=pstr+1**). This will continue till **pstr** reaches the address **3007** (see diagram), when ***pstr** becomes equal to '\0' and hence the condition becomes false. No further increment of **pstr** takes place and the loop terminates.

In line-9, the length of the string is finally calculated. Note that **str** points to **3000** and **pstr** points to **3007** after the **while** loop stops. The difference of **pstr** and **str** equals **(3007 – 3000) = 7**. That is exactly the length of the word "**Kolkata**" as displayed in line-10 using the **printf()** statement.

Important: Though the array name **str** behaves like a pointer, but we cannot use **str++** to point to the next character, as the **starting address indicated by the array name cannot be changed**.

C to it that .....

The following examples demonstrate the areas where mistakes can occur.

The following program is used to print two strings in alphabetical order.

```
1  void main()
2    { char str1[80], str2[80], temp[80];
3      printf("\nEnter first string: ");
4      gets(str1);
5      printf("\nEnter second string: ");
6      gets(str2);
7      printf("\nThe entered strings are: %s %s", str1, str2);
8      if( str1 > str2 )
9        {temp = str1;
10        str1 = str2;
11        str2 = temp;
12       }
13     printf("\nThe strings in alphabetical order are: %s, %s", str1, str2);
14   }
```

Output

```
Lvalue required
```

When the above program is compiled, instead of compiling properly, the program gives the error message, "**Lvalue required**". The reason is the incorrect way of copying the strings in lines 9 to 11. Strings cannot be copied in this manner and neither can these be compared as shown in line-8. We have to use the string handling functions to do these as shown below in the corrected lines:

```
8      if( strcmp(str1, str2) > 0 )
9        { strcpy(temp, str1);
10         strcpy(str1, str2);
11         strcpy(str2, temp);
12       }
```

Output

```
Enter first string: Pradosh
Enter second string: Mitter
The entered strings are: Pradosh Mitter
The strings in alphabetical order are: Mitter Pradosh
```

The next program compares two strings.

```
1  void main()
2    { char unit[5] = {'m', 'e', 't', 'e', 'r'}, input[10];
3      printf("\nEnter unit (meter or km): ");
4      gets(input);
5      if( strcmp(input, unit) == 0 )
6        printf("\nEntered unit is meter");
7      else
8        printf("\nEntered unit is NOT meter");
9    }
```

Output

```
Enter unit (meter or km): meter
Entered unit is NOT meter
```

In the above program, the character array `unit[]` is initialised with the string "`meter`" in line-2. Though the string entered into `input[]` in line-4 is also "`meter`" (as per the above example) but still the `printf()` function of line-8 gets executed, showing that the `if` condition in line-5 is false.

The reason is the wrong way of initialising the character array `unit[]`. If a string type data is used to initialise a character array then never forget to add the null character '\0' at the end of the string as shown in the rectified program below. Moreover increase the size of the array `unit[]` to hold 6 characters including '\0', instead of 5 characters. With the program written as above, the `strcmp()` function was unable to find the length of the string in the absence of the null character to terminate the string.

```
1  void main()
2    { char unit[6] = {'m', 'e', 't', 'e', 'r', '\0'}, input[10];
3      printf("\nEnter unit (meter or km): ");
4      gets(input);
5      if( strcmp(input, unit) == 0 )
6        printf("\nEntered unit is meter");
7      else
8        printf("\nEntered unit is NOT meter");
9    }
```

Output

```
Enter unit (meter or km): meter
Entered unit is meter
```



The Fact File

- **A string is a collection of characters to form a word, name, sentence, paragraph etc.**
- C does not provide us with any special data type to represent strings
- **A string is treated in C as a collection of characters in a character array, with a special marker to denote the end of a particular string**
- The following line of code can be used to declare a string: char str[10];
- **Since a character occupies 1 byte, thus 1 byte will be allocated for storing each element of a string array**
- Just like initialising an integer or a float array, a character array can also be initialised. Each character is enclosed within a pair of single quotes as: char str[10]={'c', 'o', 'm', 'p', 'u', 't', 'e', 'r', '\0'};
- **You cannot initialise a string after you have declared it. You can only initialise a string during the time of declaring the string array. However you can input a string anywhere you like.**
- The character '\0' though looks like two different characters but C treats it as a single character called a 'null indicator' and is used to mark the end of a string. (Note '\0' is enclosed within single quotes)
- **While declaring a character array for holding strings, always remember to keep one character extra to include the null terminator**
- If a string consists of n characters, always declare the array to consist of n+1 characters minimum, as str[n+1] and not as str[n] to keep one character extra to include the null terminator
- **Since an array of characters has a special place in C, it can be also initialised in a more convenient manner as: char str[10]= "Sayantan";**
- No address operator '&' is to be used in the scanf( ) function before the variable name to enter a string
- **The %s specifier automatically inserts the null indicator '\0' after the entered string when using scanf( )**
- The %s specifier will read only up to the character before the first blank appears. Thus if someone enters the name "Rohan Dey" then only "Rohan" will get stored in the array using scanf( ) and %s
- **The specifier %[^\n] indicates that the function will take in all characters, (including white-space characters like blank, tab etc.) until it gets the new-line character '\n' i.e. it will input all characters which are not equal to '\n' and can be used with scanf( ) to enter multiple words**
- Another form of scanf() can be used to restrict input to only lowercase and uppercase alphabets. The function stops taking a character when it gets a character other than the alphabets, as shown: scanf("%[A-Z a-z]", str);
- **The gets() function can be used to input a multiple word string**
- Use the function fflush(stdin) before gets() to enter a string
- **A character matrix can be used to store multiple strings. The first index of such a matrix defines the number of strings and the second index indicates the maximum number of characters in each string**
- strlen() function is used to count the number of characters in a string, excluding the null terminator
- **strcpy() function is used to copy one string into another (you cannot copy one string into another simply by equating them like other type of variables using the equal sign)**

- **strcmp() function is used to compare two strings. By comparing two strings is to see if two strings are identical or if they are in proper alphabetical order or not**
- strcat() function is used to join two strings i.e. one string gets appended or joined at the end of the other string

Review Questions



Q1. **Multiple Choice Questions. Select from any one of the four options.** **1 each**

i) Which of the following is an incorrect statement for a string data type in C:
   a. A string is a collection of alphabets
   b. A string can be initialised during declaration
   c. A string can contain blank spaces
   d. The scanf( ) function can be used to input a string

ii) Which of the following is an incorrect statement for a string:
   a. A string should be terminated by a null character.
   b. A string can be copied using the assignment operator '='.
   c. Strings cannot be compared by using relational operators.
   d. A character pointer can be used to represent a constant string.

iii) A string is terminated by a _____ character:
   a. '\n' b. '\r' c. '\0' d. '\b'

iv) To store a string 10 characters long you require a minimum of _____ bytes.
   a. 20 b. 22 c. 10 d. 11

v) Which of the following codes can be used to initialise a character array?
   a. char test[ ] = 'Test';
   b. char test[ ] = "Test";
   c. char test[ ] = ['T', 'e', 's', 't'];
   d. char test[ ] = {"T", "e", "s", "t", "\0"};

vi) Which of the following codes is correct for a string type data in C?
   a. char str1[10], str2[10]; gets(str1); str2=str1;
   b. char str1[10], str2[10]; scanf("%s", &str1); str2=str1;
   c. char str1[10], str2[10]; gets(str1); while (*str1 != '\0') {*str2=*str1; str1++; str2++;}
   d. char str1[10], str2[10]; gets(str1); strcpy(str2,str1);

vii) Which of the following format specifiers can be used to input multiple words using scanf( )?
   a. %^ [\n] b. ^[%\n] c. %[^\n] d. [%^\n]

viii) Which of the following functions can be used to input multiple words from a string?
   a. putchar( ) b. getchar( ) c. puts( ) d. gets( )

ix) Which of the following codes is correct:
   a. char str[10]="ABCD"; while(*str != '\0') {printf("%c", *str); str++;}
   b. char str[10]="ABCD"; int i=0; while(*(str+i) != '\0') {printf("%c", *(str+i)); i++;}
   c. char str[10]="ABCD"; while(str != '\0') {printf("%c", *str); str++;}
   d. char str[10]="ABCD"; int i=0; while(str+i != '\0') {printf("%c", str+i); i++;}

x) Which of the following is not a proper library function to work with string type data?
   a. strcpy b. strlen c. strcpm d. strcat

xi) Which of the following codes is correct in C?
   a. char X[9]="COMPUTER", Y[9]; printf("%d", strlen(X));
   b. char X[9]="COMPUTER", Y[9]; strcpy(X, Y); printf("%s", Y);
   c. char X[9]="COMPUTER", Y[9]; strcat(X, Y); printf("%s", X);
   d. char X[9]="COMPUTER", Y[9]; printf("%d", strcmp(X,Y));

xii) Which of the following library functions can be used to join two strings in C?
a. strcmp b. strcpm c. strpmc d. strccm

xiii) How many different strings can be stored using the following matrix: char list[10][8];
a. 10 b. 9 c. 8 d. 7

xiv) What is the maximum length of each string that can be stored using the matrix: char list[10][8];
a. 10 b. 9 c. 8 d. 7

xv) What is the output of the following program piece in C?

```
char str[10]="RSTUVWXYZ"; int i;
for(i=1; i<10; i+=2) printf("%c", str[i]);
```

a. TVXZ b. RTVX c. SUWY d. STUV

xvi) What is the output of the following program piece in C?

```
char x[10]="COMP", Y[10]="UTER";
strcat(x,y);
printf("%s%s", x, y);
```

a. COMPUTER b. COMPUTERUTER c. COMPUTERCOMP d. UTERCOMP

xvii) What is the output of the following program piece in C?

```
char x[10]="beginning"; int i;
for(i=0; x[i]!='\0' && x[i]!='n' && x[i]!='i'; i++)
printf("%c", x[i]);
```

a. innin b. begiig c. begg d. beg

xviii) What is the output of the following program piece in C?

```
char x[10]="program"; int i;
for(i=strlen(x)-1; i>0; i--) printf("%c", x[i]);
```

a. margor b. margorp c. argorp d. progra



Q2. **Short Answer type questions:** **1 each**

i) What do you mean by a string in C?
ii) What is the use of the '\0' character in a string in C?
iii) State one difference between a normal character array and a string array.
iv) What is the byte size of an array that can store a string with string length 10?
v) What is the use of the strcat() function?
vi) What is the use of the strcpy() function?
vii) What is the use of the strlen() function?
viii) What is the use of the strcmp() function?
ix) What type of data can be used to store a list of strings in C?
x) Maximum how many different strings can be stored using the matrix: *char x[5][10]*?
xi) Declare a matrix that can store 10 different words with a maximum of 20 characters per word.
xii) State one purpose of using a string pointer.



Q3. **Long Answer type questions:** **7 each**

i) Write a program to check if a string input by the user is a palindrome or not. Explain the use of the strcpy() function with the help of an example. 4+2+1

ii) Explain the working of the strcmp() function with the help of an example. Explain how you can store a list of words in C? What is the purpose of using the null character in a string? 4+2+1

iii) Write a program to find the length of a string without using any library function. Explain the difference between a normal character array and a string array. What is the use of the strcat() function? 4+2+1



Q4. **Assignment Programs:**

i) Write a program to input a string and print it in reverse order.

ii) Write a program to input a string and print every alternate character from the string starting from the first character.

iii) Write a program to input a sentence and then replace every blank in the sentence with the character "#". Print the final string.

iv) Write a program to input a word and convert it to another word such that every alphabet in the new word is one position ahead of the original alphabet [for example "zebra" will become "afcsb"]

v) Write a program to input a word and capitalise every alternate letter in the word starting from the first letter. You should keep a check to see if the letter which needs to be case changed is already in the required case or not.

vi) Modify the previous program by writing two functions called upper() and lower(). The first one converts any letter to uppercase irrespective of its original case and the second converts any letter to lowercase irrespective of its original case. Now call these functions to convert the alternate characters to upper and lowercase for a given word entered by the user [for example for the input word "Olympics", the output will be "OlYmPiCs"].

vii) Modify the code in the previous program to input a sentence and capitalise every alternate letter in the sentence [for example "Good morning India" will become "GoOd MoRnInG iNdIa"]

viii) Write a program to input a string and actually reverse it and then display the reversed string.

ix) Write a program to enter a sentence and capitalise the first and the last letter of each word in the sentence [for example for the input string "The world is a stage", the output should be "ThE WorlD IS A StagE"].

x) Write a program to input a word into a string type variable and then create a new string from the input word such that the new string does not have any vowel in it [for example if the input string is "intelligence", then the output string will be "ntllgnc"].

xi) Write a program to use a function to find the length of a string passed to it and return the length. Do not use the strlen() function in your program.

xii) Write a program to use a user defined function called ***copy*** with two string arguments. The function copies the content of the second string argument to the first string argument. The program inputs a string into one of the string variables and calls the ***copy*** function to make the copy. Print both the original and the copied strings after the function is exit. Do not use the strcpy() function.

xiii) Write a program to enter two strings of equal length. Find the number of places where they have the same character and the number of positions where they differ. (Convert all lowercase letters to uppercase in the two strings first, before checking)

xiv) Write a program to input a word and form a new word in a new string by removing the first and the last letters from the word [for example for the input word "computer", the output will be "ompute"].

xv) Modify the above program to use only the input character array for the conversion and no other additional array.

xvi) Write a program to input a string and remove every alternate letter from the string starting from the second character from the left and print the final string [e.g. "Sunday" will become "Sna"].

xvii) Write a program to read 10 words into a character matrix and then calculate and print the length of each string stored in the matrix. [Hint: Write a string length finding function and use it to find the length of each string as you scan the words]

xviii) Write a program to read a set of 10 names. Then print out only those names whose name starts with a particular character as input by the user.

For example, for the names – Sarmilee, Souvik, Ritwik, Rajat, Anasua, Bibek, Chandan, Sreshtha etc. – if the user enters the character 'S' to find names starting with 'S', then the program will print the names - Sarmilee, Souvik, Sreshtha from the above list.

(Hint: Store the names in a 2D character array like names[10][20]. Then use a for loop to read each name and compare the first character of each name i.e. names[i][0] with the user entered character stored in a character variable. If it matches then print the name)

xix) Write a program to read a set of 10 names and their ages. Then print out only those names whose ages are more than or equal to 50.

(Hint: Store the names in a 2D character array like names[10][20]. Store the ages in another 1D integer array like age[10]. Then use a for loop to read each age and check if it is greater than or equal to 50. If so, then print the name corresponding to the index of this age. Thus if age[i]>=50, then print names[i].)

xx) Write a program to enter the names of the week (Mon, Tue, Wed, Thu, Fri, Sat, Sun) into a string matrix. Next store the names of the week at the odd position (Mon, Wed, Fri, Sun) in a matrix called Odd and store the names of the week at the even position (Tue, Thu, Sat) in another matrix called Even.

xxi) Write a program to store a list of 10 names in a character matrix. Next sort the names in ascending order of the last letter in each string. Print the final sorted list.

For example if the list of names is ["Sabuj", "Abhrojeet", "Sayantan", "Rohan", "Sakib", "Promit", "Vikas", "Rakhi", "Sritama", "Saptarshi"], then the sorted list will be ["Sritama", "Sakib", "Rakhi", "Saptarshi", "Sabuj", "Sayantan", "Rohan", "Vikas", "Abhrojeet", "Promit"].

xxii) Write a program to store a list of 10 names in a character matrix. Next print those names (can be one or more) from the list which have the maximum length.

xxiii) Write a program to store a list of 10 words in a character matrix. Next print those words (can be one or more) from the list which have the maximum number of vowels in them.

xxiv) Write a program to input a string and a rotation factor r (a positive or negative integer). If r is positive then rotate the string to the right (clockwise manner) by r characters such that the characters from the end get joined at the beginning one by one. Similarly if r is negative then rotate the string to the left (anti-clockwise manner) by r characters such that the characters from the beginning get joined at the end one by one. Print the final string. The value of r can be more than the length of the string [for example if string is "rotation" and rotation factor is +3 then the final string will be "ionrotat". If rotation factor is -3, then the final string will be "ationrot"].

CHAPTER 15
Structures and Unions in C

- What is a structure? 15-1
- How to declare a structure 15-2
- How to Enter and Read Data from a Structure 15-4
- Array of Structures 15-5
- Passing Structures to Functions 15-8
- Union 15-9
- Some worked out problems 15-11

15.1 What is a Structure?

Till now we have learnt basically two types of variable declarations. First, a single value stored in a single variable and second, a set of values of a particular data type stored in a single variable. The second type is nothing but an array. We could have been satisfied with an array but the **major problem with an array is that it can store values of a single type only** i.e. either all elements in the array should be `int` or `float` or `char` etc.

But in general **an object may need a variety of data types to describe it.** For example to keep record of a student in a school we may require the following parameters for each student:

Name: A string type data (basically a `char` array) like "`Trinanjan Batabyal`"
Gender: A `char` type data i.e. '`M`' (for male) or '`F`' (for female)
Class: An `int` type data like `11`
Section: A `char` type data like '`B`'

It is always convenient to store the above set of data for a given student under a single variable name such as `student`, instead of using separate variables for each of the above parameters. Also to keep a record of say 'n' number of student type data in the school, we can store 'n' sets of the above type of data using an array. All these can be achieved by using a special data type called a **Structure**.

A **structure** is a **group of data** of **same or different types** but identified by a single name, thus forming a user defined data type.

A structure is a group of data of same or different types but identified by a single name.

15.2 How to declare a structure

To declare a structure one has to first **define the contents** of the structure. This will then be used as a **template** or blueprint to declare variables of that particular type. A structure for our above example is defined on the right (see box). A structure is normally **defined outside all function definitions.**

Definition of structure

To define a structure, the **keyword** `struct` is to be used along with a specific name for the new data type. The definition shown here indicates that we have defined a **structure data type** with the specific name `student`, which requires 4 different types of data to describe it. Therefore each `student` variable will require the following sub-variables to describe it:

```
struct student
 { char name[20];
   char gender;
   int clas;
   char section;
 };
```

General format

```
struct tag_name
 { type member_1;
   type member_2;
   ..............;
   type member_n;
 };
```

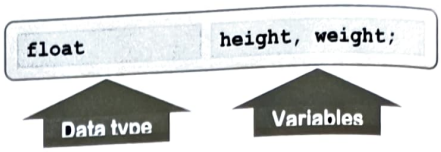
Example of a structure type data

`char name[20];` → A character **array** with 20 characters to hold the name of the student
`char gender;` → `char` type data to hold the gender of the student
`int clas;` → `int` type data storing class of the student (**class** is a keyword, hence **clas** used)
`char section;` → `char` type data to store the section of the student

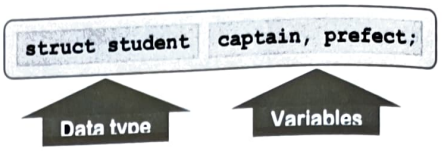
Normally a structure is defined outside all function definitions before main() function.

Each such **sub-variable** that makes up a structure is called a **member, element** or **field** of the structure. Usually the members of a structure are logically related i.e. they represent different **components or parameters to describe a particular type of object.** Note that the structure definition is terminated by a semicolon, indicating that a structure definition is also a statement.

At this point no variable has actually been created but **only** the form of the data has been defined. Once the structure data type `student` has been defined, **variables of the type `student`** can be declared just like any other variable declaration as shown below:

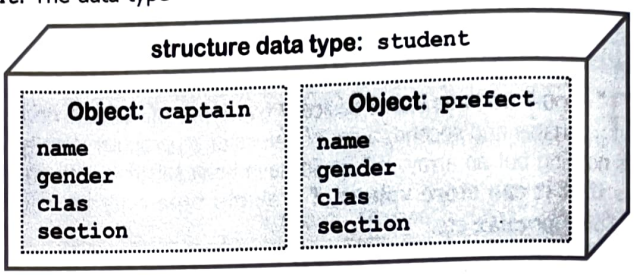



```
struct student captain, prefect;
```



Declaring structure type variable

Similar to declaration of 2 `float` type variables viz. `height` and `weight` (see example on the right), the above example declares 2 variables of type `struct student`. The data type `struct student` serves as a **template** or tag name for the new data type and the variables `captain` and `prefect` serve as different objects of type `struct student`, as shown in the figure on the right. Each of the student objects will consist of the same set of members as defined in the template.



Different ways of declaring structure variables

- **Other ways of declaring structure variables:**

Apart from the above declaration, the `struct student` type structure variables can also be declared during the structure definition. The format shown below **combines a structure definition and variable declaration of the same type in a single statement**. In this case, when the structure variables are declared along with the structure definition, the template name is optional. Thus all the three types of **declarations shown below are same**:

Same As

```
struct student
{  char name[20];
   char gender;
   int clas;
   char section;
};
struct student captain, prefect;
```

Same As

```
struct student
{ char name[20];
  char gender;
  int clas;
  char section;
} captain, prefect;
```

```
struct
{  char name[20];
   char gender;
   int clas;
   char section;
} captain, prefect;
```

A structure can be defined within the main( ) function also. In that case the definition cannot be used outside main( ).

In the **first example**, the structure is defined first and then variables `captain` and `prefect` are declared later using the newly created data type `struct student`. In the **second example**, the variables `captain` and `prefect` are declared along with the structure definition. The **third example** is similar to the second one, except that the template name `student` is left out. Note that in such a case one has to declare the variables always along with the structure definition and no other similar variables can be declared later.

A structure can be used as a member of another structure

- **Structures as structure members:**

Just like any other variable type, a structure itself can be a member of another structure, but **the member structure needs to be defined prior to its use in another structure**. Such a structure is known as a **nested structure**.

```
struct date
{ int day;
  int month;
  int year;
};
```

```
struct student
{ char name[20];
  char gender;
  int clas;
  char section;
  struct date DOB;
} captain, prefect;
```

Nested structures

The example on the right defines two types of structures. The first structure is having the specific name `date` and is used as a nested structure within the second structure called `student` to store the date of birth (DOB) of a student. Finally variables `captain` and `prefect` of type `student` are declared.

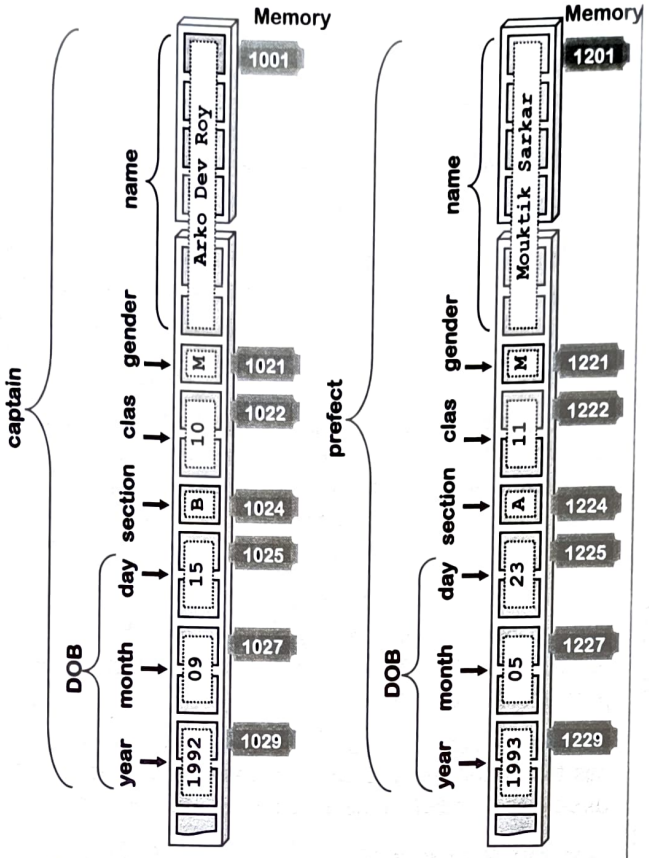
A structure must be defined before it is used as a member of another structure

The memory distribution for the two structure type variables `captain` and `prefect` declared before is shown on the next page. Note that for each `struct student` type variable, a total of 30 consecutive bytes are allocated.

The first 20 bytes hold the **name** member of the variable, the next one the **gender**, the next two bytes the **clas** and so on.

Therefore in this example we find that the memory locations 1001 to 1030 i.e. a total of 30 bytes are reserved for the variable **captain**, and the memory locations 1201 to 1230 are reserved for the variable **prefect**.

Also note that though all the members of a particular structure type variable occupy consecutive positions, the two different variables **captain** and **prefect** may not occupy consecutive locations in memory.



Memory allocation for two variables of type struct student

Members in a structure type data occupy consecutive memory locations

• Use of typedef:

Using typedef command

The keyword **typedef** can be used to make it easier to declare a structure type variable. We have seen in the last page that in C the keyword **struct** needs to be used along with the template name to declare a structure type variable like:

```
struct student captain, prefect;
```

In the above example two variables **captain** and **prefect** are declared with data type as **struct student**. Every time a variable is declared of type **student**, the keyword **struct** is to be used along with the type name.

We can overcome this requirement by using the **typedef** keyword to define the structure type data using a given type-defined name. For example consider the **student** type structure defined in the previous page. We had to write the following to declare the variables **captain** and **prefect** of type **struct student**:

```
1  struct student
2   { char name[20];
3     char gender;
4     int clas;
5     char section;
6   };
7  struct student captain, prefect;
```

However, using the **typedef** keyword, we can do the same thing as shown below, but without using the **struct** keyword while declaring the variables. The user defined word **STUDENT** is used to represent the definition of the structure. Thus instead of writing **struct student captain** etc. we can simply write **STUDENT captain** etc. as shown below:

```
1  typedef struct student
2   { char name[20];
3     char gender;
4     int clas;
5     char section;
6   } STUDENT;
7  STUDENT captain, prefect;
```

This whole definition is represented by the type defined name STUDENT.

Variables captain and prefect of type STUDENT are declared.

The typedef command can be used to define a new name for an existing data type

In the above code the type defined name **STUDENT** is used to represent the data type **struct student**. By convension a typedefined name is written in all capitals (remember in C **student** and **STUDENT** are treated as different identifier names).

Entering data into a structure

15.3 How to Enter and Read Data from a Structure

Once a structure has been defined and variables of the same type have been declared, the next step is to access the different structure members to **input data** and subsequently to read or manipulate the data stored in the different structure variables. Data can be input either by **initialising** the structure variable with some initial values or by **entering the data** during runtime.

- **Initialising a structure:**

Initialising a structure is similar to initialising an array. The data is put inside curly braces separated by commas. The following example initialises a `student` type structure as defined earlier:

Initialising a Structure

```
struct student captain = {"Arko Dev Roy" , 'M' , 10 , 'B' , 15 , 09 , 1992};
```

data type — variable — name — gender — class — section — day — month — year

An **individual structure member can be initialised separately** also. This operation can be achieved by using the '**dot**' i.e. '.' operator. The following statement assigns initial value of `10` and `'B'` to the `clas` and `section` members of the variable `captain` using the **dot** operator. The general format for the dot operator is **Variable_Name . Member_Name** as shown below:

The dot operator is used to access an element of a structure as: variable . member

```
captain.clas = 10;
captain.section = 'B';
```

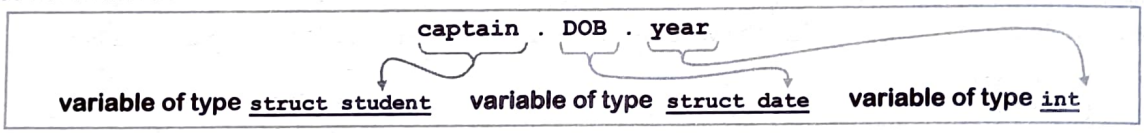
The variable or object name `captain` followed by a **dot** and then by the member name `clas` refers to that particular member of the object. Note that the member `section` being a `char` type variable, the value `'B'` is put within single quotes.

However, to access the sub-member of a structure which is itself a structure, like the date type variable, one has to **repeatedly use the 'dot' operator**. Thus to assign values to different members of the `struct date` type data `DOB` inside the `struct student` type variable `captain`, one has to write:

```
captain.DOB.day = 15;
captain.DOB.month = 09;
captain.DOB.year = 1992;
```

(**NOT** `student.date.day = 15;`)
(**NOT** `student.date.month = 09;`)
(**NOT** `student.date.year = 1992;`)

Thus the format is:



Copying a structure

- **Copying a structure / structure members:**

Similar to assigning a particular value to a member of a structure, one can also assign the value in a **member of one structure to another member of another structure**. In the example shown below, we are copying member by member the contents of the `student` type structure variable `captain`, which we have already initialised earlier, to another `student` type structure variable called `ClassMonitor`:

```
struct student ClassMonitor;
ClassMonitor.name = captain.name;
ClassMonitor.clas = captain.clas;
ClassMonitor.section = captain.section;
ClassMonitor.gender = captain.gender;
ClassMonitor.DOB.day = captain.DOB.day;
ClassMonitor.DOB.month = captain.DOB.month;
ClassMonitor.DOB.year = captain.DOB.year;
```

A structure can be copied member wise or as a whole

In the first line we have declared the `struct student` type variable `ClassMonitor`. After execution of the above statements, the value in each member of the `captain` variable will be copied to each corresponding member of the `ClassMonitor` variable.

However unlike an array, **a whole structure variable can be assigned to another structure variable.** The value stored in each member of one of the structure variables will be automatically assigned to the corresponding members of the other structure variable. But in this case the **two structures should be of**

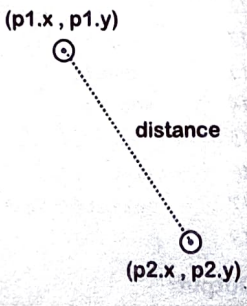
the same type. Therefore the above job can be done by simply using the assignment statement as shown below, by equating the whole structure variable to another structure variable:

```
struct student ClassMonitor;
ClassMonitor = captain;
```

Entering values into a structure during runtime:

The following program exhibits how one can enter the data into a structure at runtime using the same technique as a normal variable. The program defines a structure type data called point, where each point type data contains an 'x' component and a 'y' component. The program then enters data for two different points and calculates the distance between those points.

```
1  /*Program-143: To find the distance between two points using structure type data*/
2  #include<stdio.h>
3  #include<math.h>
4  struct point
5   {int x;
6    int y;
7   };
8  void main()
9  {struct point p1, p2;
10  float distance;
11  int temp;
12  printf("\nEnter x coordinate of point1: "); scanf("%d", &p1.x);
13  printf("\nEnter y coordinate of point1: "); scanf("%d", &p1.y);
14  printf("\nEnter x coordinate of point2: "); scanf("%d", &p2.x);
15  printf("\nEnter y coordinate of point2: "); scanf("%d", &p2.y);
16  temp = pow((p1.x - p2.x),2) + pow((p1.y - p2.y),2);
17  distance = sqrt(temp);
18  printf("\nThe distance between the two points is %f", distance);
19 }
```



Output:

```
Enter x coordinate of point1: 5
Enter y coordinate of point1: 10
Enter x coordinate of point2: 15
Enter y coordinate of point2: 3
The distance between the two points is 7.141428
```

In lines-4 to 7, the **struct point** type data is defined to have two members **x** and **y** of type **int**. In line-9 two **struct point** type variables namely **p1** and **p2** are declared. The values of the **x** and **y** components of **p1** are entered in lines-12 and 13, while the values of the **x** and **y** components of **p2** are entered in lines-14 and 15 respectively. In line-16, the value $(p1.x - p2.x)^2 - (p1.y - p2.y)^2$ is calculated using the pow() function and the value stored in the variable **temp**. Finally in line-17, the distance is calculated by taking the square root of **temp** and the result printed in line-18.

Entering values into a structure during run-time

A structure type variable can be copied to another structure type variable using the assignment operator.

The individual structure members are input separately to input data into a structure type variable.

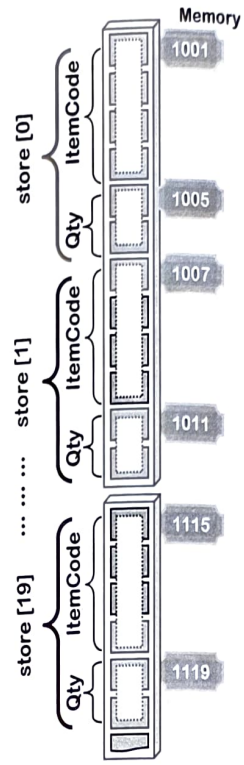
15.4 Array of Structures

Just as arrays of any particular data type can be declared, one can also declare arrays that contain objects of a particular structure type variable. The following declaration, declares an array variable called **store**, containing 20 elements of structure type **item** as defined below. The structure **item** has two components – an item code member **ItemCode** which is a 4 byte string and stores the code of a particular item in the store and a quantity member **Qty** which stores the total quantity of that particular item available in the store.

```
struct item
 {char ItemCode[4];
  int Qty;
 };
struct item store[20];
```

Array of structures

The diagram on the right shows how the array elements are stored in memory for the array `store[20]`, each element of which is a structure of type `item`. It can be seen that the members of each structure data are stored in **consecutive memory locations** for each of the array elements. Moreover the array elements themselves are also stored in consecutive memory locations i.e. `store[0]` is stored in location 1001 to 1006, `store[1]` is stored in location 1007 to 1012 etc.

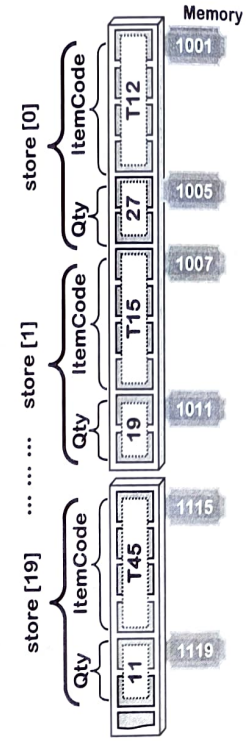


The example given below illustrates how to enter array elements for the structure array `store[]` and then display the stored data.

```
1  /*Program-144: Store Items using structure type data array*/
2  #include<stdio.h>
3  struct item
4   {char ItemCode[4];
5    int Qty;
6   };
7  void main()
8  {struct item store[20];
9   int i;
10  for(i=0; i<20; i++)
11    {printf("\nEnter Item Code-%d (3 letter code): ", i);
12     gets(store[i].ItemCode);
13     printf("\nEnter the quantity of the Item Code-%d: ", i);
14     scanf("%d", &store[i].Qty);
15    }
16  for(i=0; i<20; i++)
17     printf("\nItem Code-%s, Quantity=%d", store[i].ItemCode, store[i].Qty);
18 }
```

Output:

```
Enter Item Code-0 (3 letter code): T12
Enter the quantity of the Item Code-0: 27
Enter Item Code-1 (3 letter code): T15
Enter the quantity of the Item Code-0: 19
... ... ...
Enter Item Code-19 (3 letter code): T45
Enter the quantity of the Item Code-0: 11
Item Code-T12, Quantity = 27
Item Code-T15, Quantity = 19
... ... ...
Item Code-T45, Quantity = 11
```



The meaning of the statements in lines-12 and 14 are given below:

```
store[i].ItemCode
```

Array element **i** — Structure member `ItemCode` for that array element **i**

```
store[i].Qty
```

Array element **i** — Structure member `Qty` for that array element **i**

Therefore the statement `store[3].ItemCode` indicates the Item Code of the array element 3. Note that in line-12, the `gets()` function is used to enter string value of the Item Code and hence the '`&`' is not used, while in line-14 since the data entered is an integer, the '`&`' is used with `store[i].Qty`.

Matrix of structure type data:

Similar to an array, one can declare a matrix consisting of structure members. The following example declares a structure called COLOUR that stores the different parameters of a CRT monitor screen pixel as a matrix, where the (x, y) position of a pixel is given by the values HORI and VERT:

2D array of structure type data

```
struct COLOUR
{ int red;
  int green;
  int blue;
  int hue;
  int saturation;
  int luminosity;
} pixel[HORI][VERT];
```

Structure containing arrays:

We have seen that the assignment operator can be used to copy an entire structure variable without the need to copy the compenents separately. We can use this property of a structure to copy a string or array embedded inside a structure as shown below:

Structure with arrays

```
1  /*Program-145: Using a structure with array component*/
2  #include<stdio.h>
3  struct array
4   {int arr[5];};
5  void main()
6  {struct array x, y;
7   int i;
8   for(i=0; i<5; i++)
9     {printf("\nEnter value[%d] in x: ", i);
10     scanf("%d", &x.arr[i]);
11    }
12  y = x;
13  for(i=0; i<5; i++)
14     printf("\n%d in y", y.arr[i]);
15 }
```

A structure can be used to store array type data. Such arrays can be copied using the assignment operator =

Output:

```
Enter value[0] in x: 4
Enter value[1] in x: 6
Enter value[2] in x: 9
Enter value[3] in x: 4
Enter value[4] in x: 1
4 in y
6 in y
9 in y
4 in y
1 in y
```

The structure **array** is defined in lines-3 and 4. It contains a single integer array type component **arr[5]**. Two **struct array** type data **x** and **y** are declared in line-6. The **for** loop of line-8 is used to input values in the variable **x**. The **scanf()** function of line-8 is used to input the array element using the variable **x** and the dot operator to access the array component **arr[i]**.

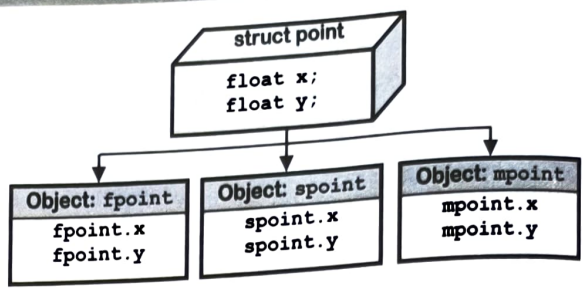
In line-12 the structure variable **x** is copied to the structure variable **y**. In doing so, the content of the entire array in **x** gets automatically copied to the array component of the structure variable **y**. When the **for** loop of line-13 prints the content of the array in the variable **y**, the same values as stored in **x** get displayed as seen in the output shown above.

15.5 Passing Structures to Functions

A structure can be passed member wise or as a whole to another function

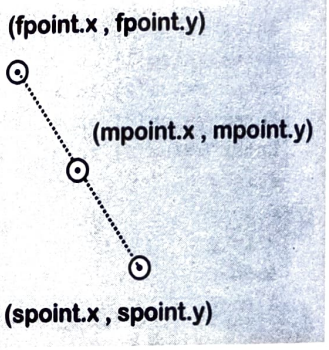
Just like a normal variable, we can pass an entire structure or members of a structure to a function **using the normal call by value method**.

The following example uses the structures called `struct point` to denote the pair of (x, y) co-ordinates of any point and finds the co-ordinates of the middle point `mpoint` of a line segment whose ends are represented by the two points `fpoint` and `spoint`:



Program to pass a structure to a function

```
1  /*Program-146: Passing Structures to Functions - To find Mid-Point of two points*/
2  #include<stdio.h>
3  struct point
4   {float x;
5    float y; };
6  struct point MakePoint( float a, float b );
7  struct point MidPoint( struct point p1, struct point p2 );
8  void main()
9   {struct point fpoint, spoint, mpoint;
10   float x1, y1;
11   printf("\nEnter the x co-ordinate of point1: "); scanf("%f", &x1);
12   printf("\nEnter the y co-ordinate of point1: "); scanf("%f", &y1);
13   fpoint = MakePoint(x1, y1);
14   printf("\nEnter the x co-ordinate of point2: "); scanf("%f", &x1);
15   printf("\nEnter the y co-ordinate of point2: "); scanf("%f", &y1);
16   spoint = MakePoint(x1, y1);
17   mpoint = MidPoint( fpoint, spoint );
18   printf("\nThe x co-ordinate of the mid point = %.2f", mpoint.x );
19   printf("\nThe y co-ordinate of the mid point = %.2f", mpoint.y );
20  }
21 struct point MakePoint( float a, float b )
22  {struct point temp;
23   temp.x = a;
24   temp.y = b;
25   return temp;
26  }
27 struct point MidPoint( struct point p1, struct point p2 )
28  {struct point temp;
29   temp.x = (p1.x + p2.x)/2.0;
30   temp.y = (p1.y + p2.y)/2.0;
31   return temp;
32  }
```



Output:

```
Enter the x co-ordinate of point1: 5.6
Enter the y co-ordinate of point1: 2.2
Enter the x co-ordinate of point2: 3.6
Enter the y co-ordinate of point2: 7.4
The x co-ordinate of the mid point = 4.60
The y co-ordinate of the mid point = 4.80
```

The above program uses a structure data type called `point`, which stores the `x` and `y` co-ordinates of a point. Note that the structure is defined outside the `main()` function from line-3 to 5, so that it becomes a **global variable type** and can be accessed by all the functions.

Within the `main()` function, three variables are declared of type `struct point`, called `fpoint`, `spoint`, and `mpoint` in line-9. The x and y components of the first point are input by the user in lines-11 and 12 within the `float` variables `x1` and `y1`. The function `MakePoint()` is then called in line-13. The purpose of the function is to accept the two values `x1` and `y1` within the variables `a` and `b` and to convert them to a single point type structure variable and return it. The function thus returns a `struct point` type data, which is assigned to `fpoint` in the `main()` function.

In lines-14 and 15, the components of the second point are again entered within the variables `x1` and `y1` and in line-16 the `makepoint()` function is again called to make the second point type data which is returned by the function and stored in the variable `spoint` in `main()`.

Finally the function `MidPoint()` is called in line-17 with the struct point type arguments `fpoint` and `spoint`. The function `MidPoint()` uses a temporary `struct point` type variable called `temp`, to calculate the co-ordinates of the middle point in lines-29 and 30. Within the function header, the formal arguments `p1` and `p2` of type `struct point` are used to receive the actual arguments `fpoint` and `spoint`. The components of the middle point are calculated within the function and returned as a `struct point` type data, to be received by the variable `mpoint` in `main()`.

Finally the x and y components of `mpoint` are displayed in `main()` as `mpoint.x` and `mpoint.y` in lines 18 and 19 respectively.

Note: The function `MidPoint()` accepts two `struct point` type data and hence the variables `p1` and `p2` are taken as `struct point` type variables. The function also returns a `struct point` type data i.e. `temp` and hence the return data type of the function is also taken as `struct point`.

15.6 Union

A **union** is another **user defined composite data type** similar to a structure and defined using the keyword `union`. However the major difference between a union and a structure is that unlike the members of a structure which occupy different storage locations, **all the members of a union share the same storage area** within the memory. Thus it can **handle only one member at a time** and is basically used to **conserve memory**. The syntax of a union is similar to that of a structure and is shown below:

```
union unit
 { int cm;
   float inch;
   char str[4];
 } ;
```

General format

```
union tag_name
 { type member_1;
   type member_2;
   ..............;
   type member_n;
 };
```

Defining a union type data

A Union is a user defined data type where all the union members occupy the same memory location.

```
union unit
   { int cm;
     float inch;
     char str[4];
   } Tshirt;
```

2001	2002	2003	2004	
				← Memory Location
				← char str[4] , 4 bytes
				← float inch , 4 bytes
				← int cm , 2 bytes

The size of a union is same as the size of the largest data member in the union.

The above definition defines a union with the template or tag name as `unit`. The members include an `int` type variable called `cm`, a `float` variable called `inch` and a string array type variable called `str[4]`. The `union unit` is used to describe a garment size in different units.

Next a variable called `Tshirt` of type `union unit` is declared. We can see from the adjacent diagram that the member variables of `union unit` are of different data types. Hence the compiler **allocates enough space to hold the largest member** of the `union`. In this case, it is the `float` member called `inch` or the string member called `str[4]` which occupies 4 bytes and hence the maximum space allotted is 4 bytes. All the three members share the same memory location starting from 2001 to 2004. However the integer requires only two bytes. Hence when the integer data is stored, the other two bytes remain unused.

Only one data member of a union type data remains active at any given time.

To access a union member the same syntax as used to access a structure member is used as shown below:

```
Tshirt.cm ← to input value 40
Tshirt.inch ← to input value 15.25
Tshirt.str ← to input value "XXL"
```

The above three statements assign different values to the different member variables. Note that the different values corresponding to a shirt size of 40cm are assigned to the different members. However, since at any given time only one of the member variables of a union can be active, **one cannot access more than one member variable at any instance of time**. This is the major difference between a structure and a union. One can initialise all structure members, but **one can access only a single union member**. Thus the following set of statements will produce garbage values.

```
printf("\nEnter shirt size in cm: ");
scanf("%d", &Tshirt.cm);
if(Tshirt.cm == 40)
   Tshirt.inch = 15.75;
printf("\nEntered shirt size = %d", Tshirt.cm);
```

In the above program piece, the value is input into the member variable `Tshirt.cm`. Next, in case the `if` statement is true the `Tshirt.inch` member is assigned the value `15.75`. This automatically erases the value of `40` from the shared memory location for `Tshirt.cm`. Hence when in the last statement `Tshirt.cm` is to be printed, it will print some garbage value as no value exists for `Tshirt.cm` at that instant.

The next program shows the **use of the above union type data**.

```
1  /*Program-147: Using union type data*/
2  #include<stdio.h>
3  union unit
4   { int cm;
5     float inch;
6     char str[4];
7   };
8  void main()
9   {union unit Tshirt;
10   int option;
11   printf("\nEnter <1> for cm <2> for inch <3> for size: "); scanf("%d", &option);
12   if (option==1)
13      { printf("\nEnter the shirt size in cm.: ");
14        scanf("%d", &Tshirt.cm);
15      }
16   else if (option==2)
17      { printf("\nEnter the shirt size in inch.: ");
18        scanf("%f", &Tshirt.inch);
19      }
20   else if (option==3)
21      { printf("\nEnter the shirt size as 'S', 'M', 'L', 'XL', 'XXL': ");
22        fflush(stdin);
23        gets(Tshirt.str);
24      }
25   else
26      printf("\nYou have not entered a valid choice...");
27   if (option==1) printf("\nThe entered Tshirt size is %d cm", Tshirt.cm);
28   if (option==2) printf("\nThe entered Tshirt size is %.2f inch", Tshirt.inch);
29   if (option==3) printf("\nThe entered Tshirt size is %s", Tshirt.str);
30  }
```

Output:

```
Enter <1> for cm <2> for inch <3> for size: 3
Enter the shirt size as 'S', 'M', 'L', 'XL', 'XXL': XL
The entered Tshirt size is XL
```

The union called unit is defined in line-3 to 7. In line-11 the option to input the size is displayed. The `scanf()` function inputs the choice. Depending upon the choice the `if` statement of line-12 or the `else` statements of line-16, 20, or 25 get executed and the input taken accordingly. Note that before using the `gets()` function the `fflush(stdin)` statement is used to clear the input buffer.

Finally the three `if` statements in line-27 to 29 are used to print the size entered.

Unions as members of other user defined data types:

Like nested structures, one can have unions with union members, structures with union members and unions with structure members. The following example declares a `union` with the tag name `unit` that is used to describe the size of a garment. Next a structure called `clothes` is defined to be used to store the data for various clothes. The `size` member of the structure is basically a `union unit` type data and is used to store the size of a particular garment in a given unit.

A Union type data can be used as a component for another structure or union definition.

```
union unit
  { int cm;
    float inch;
    char str[4];
  };
struct clothes
  {char manufacturer[20];
   float cost;
   union unit size;
  };
struct clothes Shirt, Tshirt;
```

Structure with nested Union type data

The variables `Shirt` and `Tshirt` are declared to be of type `struct clothes`. To access the `size` member of the `Tshirt` variable for example, one can write the following code:

```
Tshirt.size.inch = 15.25
```

15.6 Some worked out problems

The next program declares a book type structure to store the name, author, edition, and price of a book and a variable `b` of type `struct book`. The values within the book members are then entered using the input functions `gets()` and `scanf()`. Finally the whole data is displayed using a `printf()` statement.

Worked out problems

```
1  /*Program-148: To store the details of a book in a structure*/
2  #include<stdio.h>
3  struct book
4   {char name[20];
5    char author[20];
6    int ed;
7    float price;
8   };
9  void main()
10 {struct book b1, b2;
11  printf("\nEnter name of book1: ");
12   fflush(stdin); gets(b1.name);
13  printf("\nEnter author of book1: ");
14   fflush(stdin); gets(b1.author);
```

```
15   printf("\nEnter edition of book1: ");
16    scanf("%d", &b1.ed);
17   printf("\nEnter price of book1: ");
18    scanf("%f", &b1.price);
19   printf("\nEnter name of book2: ");
20    fflush(stdin); gets(b2.name);
21   printf("\nEnter author of book2: ");
22    fflush(stdin); gets(b2.author);
23   printf("\nEnter edition of book2: ");
24    scanf("%d", &b2.ed);
25   printf("\nEnter price of book2: ");
26    scanf("%f", &b2.price);
27   if (b1.price > b2.price)
28      printf("\nBook '%s' by '%s' is having higher price", b1.name, b1.author);
29   else
30      printf("\nBook '%s' by '%s' is having higher price", b2.name, b2.author);
31   return 0;
32  }
```

Output:

```
Enter name of book1: Professor Shonku Collection
Enter author of book1: Satyajit Ray
Enter edition of book1: 3
Enter price of book1: 450.00
Enter name of book2: Two Cities
Enter author of book2: Chetan Bhagat
Enter edition of book2: 1
Enter price of book2: 300.00
Book 'Professor Shonku Collection' by 'Satyajit Ray' is having higher price
```

Note that the `gets()` function is used to enter the strings `b1.name` and `b1.author` while `scanf()` can be used for entering the edition `b1.ed` and price `b1.price` etc. Also the `fflush(stdin)` function is used before each `gets()` statement to clear the input buffer.

The next program is used to define a structure to store the item code and quantity of different items in stock in a store and to find the item with the maximum quantity.

```
1   /*Program-149: Stores Items and Finds Maximum quantity in store*/
2   #include<stdio.h>
3   typedef struct item
4    {char ItemCode[4];
5     int Qty;
6    } ITEM;
7   int main()
8   {ITEM store[20]; int i, index, max;
9    for(i=0; i<20; i++)
10     {printf("\nEnter Item Code-%d (3 letter code): ", i);
11      fflush(stdin); gets(store[i].ItemCode);
12      printf("\nEnter the quantity of the Item Code-%d: ", i);
13      scanf("%d", &store[i].Qty);
14     }
15   max = store[0].Qty;
16   for(i=0; i<20; i++)
17       {if(store[i].Qty > max)
18           {max = store[i].Qty;
19            index = i;
20           }
21       }
```

```
22  printf("\nItem Code - %s had the maximum quantity", store[index].ItemCode);
23  return 0;
24  }
```

The above program is used to enter the item-code and quantity of a list of 20 items. In addition to that it finds the item code with the highest quantity of items in store, using the **for** loop from line 16 to 21.

The next program is used to define a structure called **STRING** using **typedef** to store and manipulate string type data.

Type-defining a STRING type structure

```
1   /*Program-150: To use a STRING type data to store strings*/
2   #include<stdio.h>
3   typedef struct string
4    {char str[80];
5    } STRING;
6   int main()
7   {STRING names[100];
8    int i, n;
9    printf("\nEnter number of names to store: ");
10   scanf("%d", &n);
11   for(i=0; i<n; i++)
12     {printf("\nEnter name[%d]: ", i);
13      fflush(stdin); gets(names[i].str);
14     }
15   for(i=0; i<n; i++)
16     {if(names[i].str[0] == 'S')
17        printf("\n%s", names[i].str);
18     }
19   return 0;
20  }
```

Output:

```
Enter number of names to store: 5
Enter name[0]: Sujoy
Enter name[1]: Debesh
Enter name[2]: Avik
Enter name[3]: Sayantan
Enter name[4]: Tathagata
Sujoy
Sayantan
```

In the above program an array of **STRING** type data called **names** is declared in line-7. The names are entered using the **for** loop of line-11. The **for** loop of line-15 then checks whether the starting letter of a name is 'S' or not using the **if** statement of line-16. If so, it prints the name in line-17.

The next program uses the **STRING** type structure defined in the previous program to exchange two strings.

Exchanging two strings

```
1   /*Program-151: To exchange two string type data*/
2   #include<stdio.h>
3   typedef struct string
4    {char str[80];} STRING;
5   int main()
6   {STRING nameA, nameB, temp;
7    printf("\nEnter first name: ");
8    fflush(stdin); gets(nameA.str);
9    printf("\nEnter second name: ");
10   fflush(stdin); gets(nameB.str);
11   printf("\nThe entered strings are: %s and %s", nameA.str, nameB.str);
12   temp = nameA;
```

```
13   nameA = nameB;
14   nameB = temp;
15   printf("\nThe exchanged strings are: %s and %s", nameA.str, nameB.str);
16   return 0;
17  }
```

Output:

```
Enter first name: Sachin
Enter second name: Sourav
The entered strings are: Sachin and Sourav
The exchanged strings are: Sourav and Sachin
```

In the above program we have used the same structure defined in the last program. Three **STRING** type data **nameA**, **nameB**, and **temp** are declared in line-6. The names are entered in lines-8 and 10 respectively. Line-11 prints the strings as entered in the variable **nameA** and **nameB**. The code in lines-12 to 14 are then used to exchange the content of the two strings using a third **STRING** type data **temp** just like normal numeric values and without using **strcpy()** function. The exchanged strings are printed in line-15.



Difference between Array and Structure

Difference between Array and Structure type data

Array	Structure
1. Can contain data of same data type only	1. Can contain different types of data
2. No reserved keyword is used to declare an array	2. The struct keyword is used to define a structure
3. Can be initialised only during declaration as int arr[4]={6,8,3,7};	3. Can be initialised during declaration and later also
4. An array cannot be directly copied into another array using the assignment operator	4. A structure type variable can be copied to another variable using the assignment operator
5. If a single element of a numeric array is initialised, the remaining array elements get automatically initialised to 0 (zero)	5. Members of a structure type variable does not get initialised automatically based on the value of some other data members
6. An array arr can be declared as: int arr[10];	6. A structure type data can be defined as: struct point {int x, y;};



Difference between Structure and Union

Difference between Structure and Union

Structure	Union
1. The keyword struct is used to define a structure	1. The keyword union is used to define a union
2. The total memory occupied by a structure type data is equal to the sum of the memory requirement of the individual components	2. The total memory requirement is equal to the memory requirement of the largest component of the union
3. All the members of a structure remain active and can be used together	3. Only one member of a union is active at a time and can be used at a given time
4. Example: struct point {int x, y;};	5. Example: union size { int cm; float inch };



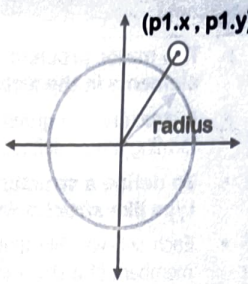
C to it that .....

The following examples demonstrate the areas where mistakes can occur.

The next program is used to find whether a point is inside, outside or on a circle using a point type structure:

```
1   /*Program-152: To find the position of a point in a circle*/
2   #include<stdio.h>
3   struct point
4    {int x;
5     int y;
6    };
```

```
7  void main()
8  {struct point p1;
9   long int radius, sq;
10  printf("\nEnter x coordinate of the point: ");
11   scanf("%d", &point.x);
12  printf("\nEnter y coordinate of the point: ");
13   scanf("%d", &point.y);
14  printf("\nEnter the radius of the circle with centre at 0,0: ");
15   scanf("%ld", &radius);
16  sq =  point.x * point.x + point.y * point.y;
17  if(sq > radius*radius)
18     printf("\nPoint is outside the circle");
19  else if(sq < radius*radius)
20     printf("\nPoint is inside the circle");
21  else if(sq == radius*radius)
22     printf("\nPoint is on the circle");
23  }
```



When the program is compiled it will not compile and the following error message will be displayed:

Output

```
Undefined symbol point
```

The reason is the **wrong use of the structure definition**. Note that in line-3, the structure is defined with the tag name as `point` and in line-8 a variable `p1` of type `struct point` is defined. Hence we have to use the variable name `p1` whenever we access the structure. However in lines-11, 13 and 16 we have **wrongly used the structure definition** tag name `point` along with the member variable names `x` and `y`. The correct use will be `p1.x` and `p1.y` instead of `point.x` and `point.y`. The rectified program is shown below:

```
1  /*Program-152a: To find the position of a point and a circle*/
2  #include<stdio.h>
3  struct point
4   {int x;
5    int y;
6   };
7  void main()
8  {struct point p1;
9   long int radius, sq;
10  printf("\nEnter x coordinate of the point: ");
11   scanf("%d", &p1.x);
12  printf("\nEnter y coordinate of the point: ");
13   scanf("%d", &p1.y);
14  printf("\nEnter the radius of the circle with centre at 0,0: ");
15   scanf("%ld", &radius);
16  sq = p1.x*p1.x+p1.y*p1.y;
17  if(sq > radius*radius)
18     printf("\nPoint is outside the circle");
19  else if(sq < radius*radius)
20     printf("\nPoint is inside the circle");
21  else if(sq == radius*radius)
22     printf("\nPoint is on the circle");
23  }
```

The Fact File

- **The major problem with an array is that it can store values of a single data type only i.e. either all elements in the array should be int or float or char etc.**
- A structure is a group of one or more variables, usually of various types, and identified by a single name, thus forming a user defined data type
- **To define a structure, the keyword struct is to be used along with a specific name for the new data type like *struct point* etc.**
- Each sub-variable that makes up a structure is called a member, element or field of the structure. Usually the members of a structure are logically related i.e. they represent different components or parameters to describe a particular type of object
- **A structure itself can be a member of another structure, but the member structure needs to be defined prior to its use in another structure. Such a structure is known as a nested structure**
- Initialising a structure is similar to initialising an array. The data is put inside curly brackets separated by commas
- **To access a structure member the 'dot' operator (.) is used. The general format for the dot operator is VariableName.MemberName**
- To access the sub-member of a structure which is itself a structure, like the date type variable, one has to repeatedly use the 'dot' operator
- **Unlike an array, a whole structure variable can be assigned to another structure variable. The value stored in each member of one of the structure variables will be automatically assigned to the corresponding members of the other structure variable**
- Just as arrays of any particular data type can be declared, one can also declare arrays that contain objects of a particular structure type variable
- **The members of each structure data are stored in consecutive memory locations for each of the array elements. Moreover the array elements themselves are also stored in consecutive memory locations**
- Just like a normal variable, we can pass an entire structure to a function using the normal call by value method
- **The keyword typedef can be used to make it easier to declare a structure type variable**
- A union is another user defined composite data type, similar to a structure and defined using the keyword union
- **The major difference between a union and a structure is that unlike the members of a structure which occupy different storage locations, all the members of a union share the same storage area within the memory. Thus it can handle only one member at a given time and is basically used to conserve memory**
- Like nested structures, one can have unions with union members, structures with union members and unions with structure members

Review Questions

Q1. **Multiple Choice Questions. Select from any one of the four options.** **1 each**

i) All members of a structure type data:

a. are of the same data type b. can be of same data type

c. are of different data types d. must be of same data type

ii) Which operator is used to access the members of a structure?

a. star b. colon c. dot d. hyphen

iii) Members of a structure:

a. always occupy consecutive memory locations

b. may not occupy consecutive memory locations

c. sometimes occupy consecutive memory locations

d. never occupy consecutive memory locations

iv) How many bytes are required to store a structure variable that can store a name up to a maximum of 20 alphabets in size and an address that can be up to a maximum of 60 alphabets in size.

a. 80 bytes b. 81 bytes c. 82 bytes d. 83 bytes

v) What will be the byte requirement for a structure type data that stores a character array of 10 characters, two float type data, and a long double type data.

a. 4 bytes b. 8 bytes c. 10 bytes d. 28 bytes

vi) What will be the byte requirement for a union type data that stores a character array of 10 characters, two float type data, and a long double type data.
a. 28 bytes b. 4 bytes c. 10 bytes d. 8 bytes

vii) What is the output of the following program piece in C?
```
struct {int x, y; float z;} a={10, 5, 2.5}, b;
b.x=a.y; b=a;
printf("\n%d", 2*(b.x – b.y)/b.z);
```
a. 0 b. 2 c. 4 d. 6

viii) What is the output of the following program piece in C?
```
struct point {int x, y;} arr[5]={1,2,3,4};
void main( )
  { printf("\n%d,%d", arr[2]); }
```
a. 1,2 b. 2,2 c. 1,1 d. 0,0

Q2. **Short Answer type questions:** **1 each**

i) What do you mean by a structure?
ii) What is the use of a structure type data?
iii) What do you mean by a structure tag name?
iv) What is the use of the dot operator in a structure?
v) Define a structure with an array member called list[] containing 100 floating point values.
vi) State one use of the keyword 'typedef'.
vii) What is the purpose of using a union type data?
viii) State one difference between a structure and a union type data.
ix) What do you mean by a nested structure?
x) What do you mean by the phrase 'an array of structures'?

Q3. **Long Answer type questions:** **7 each**

i) State any two methods of declaring structure type variables of a given structure definition. Show how you can access a structure member in a program. State two differences between a structure and an array. What is the difference between a function defined outside the main function and a function defined within the main function? 2+2+2+1

ii) Explain with the help of a proper example what you mean by a nested structure. Initialise the components of a variable w of a structure type data *struct weight { int Kg; int gm; int mg; };* with the values Kg=10, gm=25, mg=11. What do you mean by an array of structures? 4+2+1

iii) Define a structure and declare two variables to store the radius, height, and density of a solid cone type data. What is a union? What is the use of the 'typedef' keyword with respect to a structure type data? 4+2+1

Q4. **Assignment Programs:**

i) Create a structure called **struct student** to specify data of students as given below: 4
[ID, Name, Class, Sec, JoinYear]
Assuming there are not more than 100 students in the school. Write a program in C to input the student data for n such students. Next print the data of a student whose ID is entered. [Hint: Use an array of 100 structure elements]

ii) With reference to the previous problem print the names of all students who joined in a particular year. The required joining year is input by the user. 4

iii) With reference to the previous problem write a function called Find( ) to print the names of all students who joined in a particular year. The required joining year is input by the user and passed to the function Find( ) as an argument. 4

iv) Create a structure called **struct box** that stores the length, breadth, and height of a rectangular

box type data. Write a function called volMax( ) that receives two box type data and returns the volume of the box with the larger volume. Within the main function declare two box type objects and call the above defined function with these as the arguments. The larger volume returned by the function is printed in the main function. 4

v) Create a structure called **struct cone** with radius and height of the cone as components. Within main, declare an array to store 10 such cone type data. Enter parameters for the 10 cone type data. Next scan the array and print the parameters of the cone with the largest volume.

vi) Create a structure called **struct bank** to specify data of customers in a blank. The data to be stored is 'AccNo', 'Name', and 'Balance'. Assume a maximum of 100 customers in the bank. Write a function called transaction( ) for withdrawal and deposit of amount in an account. The customer-request for withdrawal or deposit is given in the form: Account no., Amount, and '1' for deposit and '0' for withdrawal. If existing balance in the account is below Rs. 1000/- the function prints "Balance is insufficient for specific withdrawal", else it will show the name and the updated current balance of the customer after the transaction. 4

vii) Define a structure called **struct point** that stores the (x,y) co-ordinates of a Cartesian point. Create an array to store 10 such point type data. Next print the points which are fully located in the second quadrant by reading the points from the array. 4

viii) Define a structure data type called **struct line** to hold the 2 components (**m** i.e. slope and **c** i.e. the y intercept) of a line. Define a structure data type called **struct point** to hold the 2 components (**x** and **y**) of a point. Input the components for 2 **line** type variables L1 and L2, and find the point of intersection of the lines. Store the point of intersection in a **point** type data and display the result. 6

ix) Define a structure data type called **struct vector** to hold the 3 components (x, y, z) of a vector. Input the components for 2 such vector type variables v1 and v2, and calculate the vector product (cross product) of those using a function called VecProduct( ). The function returns the product vector. Use another struct vector type variable called v3 to store the returned result. 6

[A vector **v**, is defined as $\vec{v} = (\hat{i} \cdot x + \hat{j} \cdot y + \hat{k} \cdot z)$, where x, y, z are the magnitudes of the three components and $\hat{i}, \hat{j}, \hat{k}$ are unit vectors in three orthogonal directions. The vector product

$\mathbf{v_3} = (\hat{i} \cdot x_3 + \hat{j} \cdot y_3 + \hat{k} \cdot z_3)$ of vectors:

$\mathbf{v_1} = (\hat{i} \cdot x_1 + \hat{j} \cdot y_1 + \hat{k} \cdot z_1)$ and

$\mathbf{v_2} = (\hat{i} \cdot x_2 + \hat{j} \cdot y_2 + \hat{k} \cdot z_2)$ is defined as $\mathbf{v_3} = \mathbf{v_1} \times \mathbf{v_2}$, where:

$\mathbf{x_3} = y_1z_2 - z_1y_2$, $\mathbf{y_3} = z_1x_2 - x_1z_2$, $\mathbf{z_3} = x_1y_2 - y_1x_2$]

Appendix-I

List of Programs in the Book

Programming in C: General Concepts

Program-01:	To find Area of a Circle	8-4
Program-01a:	Using constant in a program	8-10
Program-01b:	Using const keyword in a program	8-10

Input / Output and Simple Calculations in C

Program-02:	Use of scanf() function	9-4
Program-03:	Use of extra characters within scanf() control string	9-5
Program-04	To display the use of different format specifiers	9-6
Program-05	To display the ASCII value of a character	9-7
Program-06	To display the use of the %s specifier	9-8
Program-07	To display the use of the %o and %x specifiers	9-8
Program-08:	Use of getchar() function	9-12
Program-09:	Use of gets() function	9-13
Program-10:	To find number divisible by 5	9-15
Program-11:	Inch to cm conversion	9-19
Program-12:	To find circumference, area of a circle of radius r	9-19
Program-13:	To find surface area and volume of a sphere of radius r	9-20
Program-14:	To Find the Byte Length of different Data Types	9-20
Program-15:	To find total hour, minute, second in a given number of seconds	9-20
Program-16:	To exchange two values using third variable	9-21

Decision Making and Branching in C

Program-17:	To calculate different discounts using if-else	10-2
Program-18:	Use of only if statement	10-3
Program-19:	To check if even or odd number with single statement under if and else	10-4
Program-20:	Use of multiple single if statements	10-4
Program-21:	Use of nested if-else statement	10-5
Program-22:	Use of nested if-else statement	10-6
Program-23:	Use of multiple if-else nesting	10-7
Program-23a:	Use of if-else ladder structure	10-8
Program-24:	Use of conditions without relational operators	10-9
Program-25:	Checking multiple options without using logical OR operator	10-10
Program-25a:	Use of logical OR operator	10-11
Program-26:	Checking multiple options together without using logical AND operator	10-11
Program-26a:	Checking multiple options together using logical AND operator	10-12
Program-27:	To test multiple of 4 or 5 using logical OR	10-12
Program-28:	To show use of logical NOT operator	10-13
Program-29:	To print remarks based on marks using if-else ladder	10-14
Program-29a:	To print remarks based on marks using AND operator	10-14
Program-30:	To show the use of combined logical operators	10-15
Program-31:	To find the greater of two numbers using conditional operator	10-16
Program-32:	To display the greater of two numbers using conditional operator	10-17
Program-33:	To calculate different discounts using conditional operator	10-18
Program-34:	Conditional operator on the left side of assignment operator	10-18
Program-35:	Simple Calculator using switch-case	10-20
Program-36:	To check if 3 numbers are Pythagorean	10-22
Program-37:	To check if 3 line segments can form a triangle or not	10-22
Program-38:	To check if year is a leap year or not	10-23
Program-39:	To find the location of a Cartesian point	10-24
Program-40:	Calculating different area values using switch-case	10-25

Using Loops in C

Program-41:	To find average of three numbers	11-2
Program-41a:	Pseudocode to find average of three numbers using loop	11-3
Program-41b:	To find average of three numbers using while loop	11-7
Program-42:	Use of while loop to print from 1 to 5	11-7
Program-43:	To find the factorial of a number	11-8
Program-44:	To find the power of a number using a while loop	11-10
Program-45:	Getting the sum of the digits of a number	11-11
Program-46:	Repeating a program	11-12
Program-47:	Use of nested while loops to print pattern	11-13
Program-48:	Use of break statement	11-14
Program-49:	Example of continue statement	11-15
Program-50:	To enter and check for a valid input using do-while loop	11-16
Program-51:	To find the HCF or GCD of two numbers using do-while loop	11-17
Program-52:	To find the factorial of a number using a for loop	11-19
Program-53:	To find the sum of an AP series using a for loop	11-20
Program-54:	To find the value of e^x using a nested for loop	11-20
Program-55:	To find the value of sin(x) using a nested for loop	11-22
Program-56:	Use of comma operator in the test condition	11-23
Program-57:	Reversing a number and checking if it is a palindrome	11-24
Program-58:	Check if a number is a Facterone	11-25
Program-59:	Centigrade to Fahrenheit conversion Table	11-26
Program-60:	To print the multiples of 3	11-26
Program-61:	To check whether a number is prime	11-27
Program-62:	To print the Fibonacci Series up to n terms using a for loop	11-28
Program-63:	To print all the alphabets using continue statement	11-30
Program-64:	Using nested for loops to print pattern-1	11-30
Program-64a:	Using nested for loops to print pattern-2	11-31
Program-65:	Using nested for loops to print pattern-3	11-31
Program-66:	Beverage vending software using a do-while loop and switch-case	11-31

Functions in C

Program-67:	Example of a User Defined Function and Function Call	12-4
Program-68:	Example of a Function Call with Function Prototype	12-7
Program-69:	Example of a function returning a constant value	12-7
Program-70:	Example of a function returning a calculation	12-7
Program-71:	Example of a function returning a calculation	12-7
Program-72:	Example of a function returning a logical value	12-8
Program-73:	Example of a function returning a calculated value	12-8
Program-74:	Example of a function returning a constant character	12-8
Program-75:	Example of a function returning an input character	12-8
Program-76:	Example of a function returning maximum amongst 2 values	12-9
Program-77a:	Example of a function without an explicitly declared return data type	12-9
Program-77b:	Example of a function with a declared return data type	12-9
Program-78:	Function Call to change lower to upper character	12-9
Program-79:	Function to find and display the maximum value	12-10
Program-80:	To calculate the factorial of a number using recursion	12-11
Program-81:	To calculate the power of a number using recursion	12-11
Program-82:	Working of automatic variables	12-13
Program-83:	Working of static variables	12-14
Program-84:	Working of global variables	12-15
Program-85:	Using external functions in a program	12-16
Program-86:	Function to convert from Fahrenheit to Celsius	12-16
Program-87:	Function to calculate Factorial of a number	12-17
Program-88:	Calculation of nPr and nCr using factorial function	12-17
Program-89:	To find the sum of the digits of a number using a function	12-18

Program-90:	To check from a range of numbers, whether a number is prime or not	12-18
Program-91:	To calculate the hcf/gcd of two numbers using recursion	12-19
Program-92:	To calculate the n-th term of the Fibonacci series using recursion	12-20
Program-93:	To calculate the sum of the digits of a number using recursion	12-20
Program-94:	Function call having static variable to calculate value of e	12-21
Program-95:	Function call with numbers and operator	12-22
Program-96:	Function call to convert from decimal to binary	12-23
Program-97:	Using a function to print a double cone	12-23
Program-98:	Using function to print Pascal's Triangle	12-24

Arrays in C

Program-99:	To find the average of a set of numbers using an array	13-2
Program-100:	To find the standard deviation of a set of numbers using an array	13-3
Program-101:	To display the sum of the rows of a square matrix	13-6
Program-102:	To find the sum of the squares of numbers in an array	13-8
Program-103:	Use of Address Operator	13-9
Program-104:	Use of the indirection operator	13-10
Program-105:	Declaring and using a Pointer	13-11
Program-106:	Standard Deviation using Pointer Notation	13-14
Program-107:	To reverse the contents of an array	13-16
Program-107a:	To find the maximum from a list of numbers in an array	13-17
Program-108:	Inserting a value at k-th position	13-17
Program-109:	Deleting a value at k-th position	13-18
Program-110:	Selection Sort Method	13-19
Program-111:	Bubble Sort Method	13-21
Program-112:	Alternative Selection Sort Method	13-23
Program-113:	Deleting multiple values from an array	13-24
Program-114:	Searching for a value in an array using Linear Search Method	13-25
Program-115:	Search using Binary Search Method	13-26
Program-116:	To find the transpose of a square matrix	13-28
Program-117:	Addition of two matrices	13-30
Program-118:	To check if a matrix has a row with all 0's	13-30
Program-119:	To check if a matrix is an Identity Matrix	13-32
Program-120:	To check if a matrix is an upper diagonal matrix	13-33
Program-121:	Matrix Multiplication	13-34
Program-122:	Comparing 2 arrays using pointers	13-36

Strings in C

Program-123:	To find the length of a string	14-2
Program-124:	To print a string using scanf() without the %s format specifier	14-3
Program-125:	To count number of words in a sentence	14-3
Program-126:	To calculate length of multiple strings	14-4
Program-127:	To find the length of a string using strlen() function	14-6
Program-128:	To exchange two strings	14-6
Program-129:	Sorting a list of strings	14-7
Program-130:	Joining Strings	14-9
Program-131:	String Copy using Pointers	14-11
Program-132:	String input up to specific number of characters only	14-12
Program-133:	To count number of occurrences of the alphabet 'e' in a string	14-13
Program-134:	To count number of occurrences of any alphabet in a string	14-13
Program-135:	To count number of words in a sentence with multiple blanks	14-14
Program-136:	To count the number of vowels in a string	14-14
Program-137:	To check for a palindrome	14-15
Program-138:	To convert the first character of every word to capital	14-16
Program-139:	To find the occurrence of the word 'the' in a sentence	14-17
Program-140:	To check for a given sequence of letters in a string	14-18
Program-141:	To find the presence of a given word in a string	14-19

Program-142: String Length using Pointers 14-20

Structures and Unions in C

Program-143: To find the distance between two points using structure type data 15-5
Program-144: Store Items using structure type data array 15-6
Program-145: Using a structure with array component 15-7
Program-146: Passing Structures to Functions – To find Mid-Point of two points 15-8
Program-147: Using union type data 15-10
Program-148: To store the details of a book in a structure 15-11
Program-149: Stores Items and Finds Maximum quantity in store 15-12
Program-150: To use a STRING type data to store strings 15-13
Program-151: To exchange two string type data 15-13
Program-152a: To find the position of a point and a circle 15-15

Appendix-II: Acronyms

ACC - Accumulator	**EPROM** - Erasable and Programmable ROM
AGP - Accelerated Graphics Port	**EEPROM** - Electrically Erasable and Programmable ROM
AI - Artificial Intelligence	**GB** – Giga Byte
ALU - Arithmetic and Logic Unit	**HDD** – Hard Disk Drive
ASCC - Automatic Sequence Controlled Computer	**I-cycle** - Instruction Cycle
ANSI - American National Standards Institute	**IR** - Instruction Register
ASCII - American Standard Code for Information Interchange	**ISA Bus** - The Industry Standard Architecture Bus
BCD - Binary Coded Decimal	**KB** – Kilo Byte
BD - Blu-ray Disk	**LCD** – Liquid Crystal Display
BIOS – Basic Input Output System	**LSB** – Least Significant Bit
BPI – Bytes Per Inch	**MAR** - Memory Address Register
CAD - Computer Aided Design	**MB** – Mega Byte
CCD - Charged Coupled Device	**MCA** Bus - Micro Channel Architecture Bus
CD - Compact Disk	**MICR** – Magnetic Ink Character Reader
CD-R - Recordable media CD	**MSB** – Most Significant Bit
CD-R/W - Re-Writable media CD	**MUX** – Multiplexer
CD-ROM - Read Only Media CD	**OCR** – Optical Character Reader
CLI - Command Line Interface	**OMR** – Optical Mark Reader
CMYK - Cyan, Magenta, Yellow and Black	**PDA** - Personal Digital Assistant
CPI - Characters Per Inch	**PC** - Program Counter
CPU - Central Processing Unit	**PCI Bus** - Peripheral Component Interconnect Bus
CRT - Cathode Ray Tube	**PROM** - Programmable ROM
CU - Control Unit	**RAM** – Random Access Memory
CUI - Character User Interface	**RDRAM** - Rambus DRAM
DASD - Direct Access Storage Devices	**RGB** – Red, Green, and Blue
DDR SDRAM - Double Data Rate SD RAM	**RIMM** - Rambus In-line Memory Modules
DEMUX – Demultiplexer	**ROM** – Read Only Memory
DIMM - Dual In-line Memory Modules	**SRAM** - Static RAM
DOS - Disk Operating System	**SASD** - Sequential Access Storage Device
DRAM - Dynamic RAM	**SDRAM** - Synchronous DRAM
DSDD - Double Sided Double Density	**SIMM** - Single In-line Memory Modules
DSS - Decision Support System	**TFT** – Thin Film Transistor
DSSD - Double Sided Single Density	**TPS** - Transaction Processing System
DTP - Desk Top Publishing	**UPC** - Uniform Product Code (Barcode)
DVD - Digital Versatile Disk	**USB** – Universal Serial Bus
E-cycle - Execution Cycle	**VDU** – Video Display Unit
EDO RAM - Extended Data Out RAM	**VRAM** - Video RAM

Appendix-III: ASCII Table

Dec.	Binary	Value	Remarks
000	00000000	NUL	(Null character)
001	00000001	SOH	(Start of Header)
002	00000010	STX	(Start of Text)
003	00000011	ETX	(End of Text)
004	00000100	EOT	(End of Transmission)
005	00000101	ENQ	(Enquiry)
006	00000110	ACK	(Acknowledgment)
007	00000111	BEL	(Bell)
008	00001000	BS	(Backspace)
009	00001001	HT	(Horizontal Tab)
010	00001010	LF	(Line Feed)
011	00001011	VT	(Vertical Tab)
012	00001100	FF	(Form Feed)
013	00001101	CR	(Carriage Return)
014	00001110	SO	(Shift Out)
015	00001111	SI	(Shift In)
016	00010000	DLE	(Data Link Escape)
017	00010001	DC1	(XON) (Device Control 1)
018	00010010	DC2	(Device Control 2)
019	00010011	DC3	(XOFF)(Device Control 3)
020	00010100	DC4	(Device Control 4)
021	00010101	NAK	(Negative Ackn.)
022	00010110	SYN	(Synchronous Idle)
023	00010111	ETB	(End of Trans. Block)
024	00011000	CAN	(Cancel)
025	00011001	EM	(End of Medium)
026	00011010	SUB	(Substitute)
027	00011011	ESC	(Escape)
028	00011100	FS	(File Separator)
029	00011101	GS	(Group Separator)
030	00011110	RS	(Request to Send)
031	00011111	US	(Unit Separator)
032	00100000	SP	(Space)
033	00100001	!	(exclamation mark)
034	00100010	"	(double quote)
035	00100011	#	(number sign)
036	00100100	$	(dollar sign)
037	00100101	%	(percent)
038	00100110	&	(ampersand)
039	00100111	'	(single quote)
040	00101000	(	(left/open parenthesis)
041	00101001	)	(right/closing parenthesis)
042	00101010	*	(asterisk)
043	00101011	+	(plus)
044	00101100	,	(comma)
045	00101101	-	(minus or dash)
046	00101110	.	(dot)
047	00101111	/	(forward slash)
048	00110000	0	Start of the 10 dec. digits
049	00110001	1	
050	00110010	2	
051	00110011	3	
052	00110100	4	
053	00110101	5	
054	00110110	6	
055	00110111	7	
056	00111000	8	
057	00111001	9	
058	00111010	:	(colon)
059	00111011	;	(semi-colon)
060	00111100	<	(less than)
061	00111101	=	(equal sign)
062	00111110	>	(greater than)
063	00111111	?	(question mark)

Dec.	Binary	Value	Remarks
064	01000000	@	(AT symbol)
065	01000001	A	Start of Upper Case Alphs.
066	01000010	B	
067	01000011	C	
068	01000100	D	
069	01000101	E	
070	01000110	F	
071	01000111	G	
072	01001000	H	
073	01001001	I	
074	01001010	J	
075	01001011	K	
076	01001100	L	
077	01001101	M	
078	01001110	N	
079	01001111	O	
080	01010000	P	
081	01010001	Q	
082	01010010	R	
083	01010011	S	
084	01010100	T	
085	01010101	U	
086	01010110	V	
087	01010111	W	
088	01011000	X	
089	01011001	Y	
090	01011010	Z	
091	01011011	[	(left/opening bracket)
092	01011100	\	(back slash)
093	01011101	]	(right/closing bracket)
094	01011110	^	(caret/circumflex)
095	01011111	_	(underscore)
096	01100000	`	
097	01100001	a	Start of Lower Case Alphs.
098	01100010	b	
099	01100011	c	
100	01100100	d	
101	01100101	e	
102	01100110	f	
103	01100111	g	
104	01101000	h	
105	01101001	i	
106	01101010	j	
107	01101011	k	
108	01101100	l	
109	01101101	m	
110	01101110	n	
111	01101111	o	
112	01110000	p	
113	01110001	q	
114	01110010	r	
115	01110011	s	
116	01110100	t	
117	01110101	u	
118	01110110	v	
119	01110111	w	
120	01111000	x	
121	01111001	y	
122	01111010	z	
123	01111011	{	(left/opening brace)
124	01111100	|	(vertical bar)
125	01111101	}	(right/closing brace)
126	01111110	~	(tilde)
127	01111111	DEL	

Appendix-IV: DOS Commands

Command	Use	Format / Example
ATTRIB	It helps **to set or reset the attribute for a file** as read only, archive, hidden or as a system file	**C:\>** ATTRIB +r +a MyData.doc **C:\>** ATTRIB -r MyData.doc **C:\>** ATTRIB +h *.sys
BACKUP	It is **used to take backup copies of one or more files** from one disk to another	**C:\>** BACKUP D:\ B:/s
CHDIR or **CD**	It is used **to change from one directory to another** in DOS. Along with the command, the path and name of the directory to move to is to be typed	**C:\>** CD C:\Personal\Letter **C:\Personal\Letter>** CD.. **C:\Personal\Letter>** CD\
CHKDSK	It is used to **check a disk for errors and displays a status report**	**C:\>** CHKDSK D:/f
COPY	It is used **to copy a file** from one location to a different one or make a copy of a file with a different name at the same location. The copy command is to be followed by the source and the destination file names and file paths	**C:\>** COPY file1.txt backup.txt **C:\Personal>** COPY file2.c D:\ **C:\>** COPY file3.c C:\Personal **C:\>** COPY *.bmp C:\Personal
DATE	It is used to **display or modify the current date**. After typing the command, type a new date in the format dd-mm-yy or press Enter to keep the same date	**C:\>** DATE
DEL and **ERASE**	It is used **to delete a file**. The del command is to be followed by the path and file name to delete	**C:\>** DEL prog.c **C:\>** DEL C:\Home\Progs\Try.c **C:\>** ERASE D:\Temp\pic.bmp
DIR	It is used to **list all the files or sub directories** under a given directory. Along with the file name, the date & time of file creation and the size of the file is also displayed	**C:\>** DIR **C:\>** DIR/p **C:\>** DIR personal
DISKCOPY	It is used to **copy the complete contents of one floppy disk to another**	**C:\>** DISKCOPY A: B:
EDIT and **EDLIN**	These commands can be used **to create and edit a text file**	**C:\>** EDIT C:\Debayan.txt **C:\>** EDLIN C:\ Rikayan.txt
FORMAT	The FORMAT command in DOS is used **to lay down the pattern of tracks and sectors** onto a secondary storage device	**C:\>** FORMAT D:/q
LABEL	It is used **to create, change or delete the volume label** name of any drive	**C:\>** LABEL D: Mydisk
MKDIR or **MD**	It is used **to make a new directory** or folder in DOS. Along with the command, the path i.e. the location and name of the new directory is also to be typed	**C:\>** MD Project **C:\>** MD C:\Personal\Letter **C:\>** MKDIR D:\Temp
MOVE	It can be used **to move a file from one location to another**. It can also be used **to rename a directory**. After typing the command name, type the source file name along with the path and then type the destination path. In case you want to change the file name after moving the file, then write the new file name	**C:\>** MOVE file1.txt D:\Temp **C:\>** MOVE bd.c D:\Temp\bak.c **C:\>** MOVE C:\Temp\prg.c D:\ **C:\>** MOVE *.txt D:\Temp
RENAME or **REN**	It is used **to rename a file or a directory**. The rename command is to be followed by the path and file/directory name to be renamed and the changed file/directory name	**C:\>** RENAME prog.c final.c **C:\Personal>** REN TC.exe C.exe **C:\>** REN Personal Home
RESTORE	It is **used to get back the files**, directories and sub-directories backed up using BACKUP command. The command name is followed by the drive and path which contains the backup files and then the target drive	C:\> RESTORE B:\ D:/s/p
RMDIR or **RD**	It is used **to remove or delete a directory** or folder in DOS. Along with the command, the path i.e. the location and name of the directory to be removed is to be typed	**C:\>** RD Project **C:\>** RD C:\Personal\Letter **C:\>** RMDIR D:\Temp

Command	Use	Format / Example
SCANDISK	The SCANDISK command in DOS is used **to check the hard disk or floppy disks for logical and physical errors** and repair the problems on the disk	C:\> SCANDISK/all
TIME	It is used to **display or modify the current time.** After typing the command, type a new time in the format hour-min-sec or press Enter to keep the same time	C:\> TIME
TREE	Allows the user to view a listing of files and folders in an easy to read graphical format, with the main directory at the top, followed by the sub and sub-sub directories along its branches	C:\> TREE C:\Personal
UNDELETE	It is **used to restore a deleted file**, if possible	C:\> UNDELETE D:\project.doc
VOL	It is used **to display the disk or drive space specified**	C:\> VOL C:
XCOPY	It is used **to copy files and directory trees** from one location to another. The command will copy all the files and directories and sub-directories under it	C:\> XCOPY C:\Temp D:\Home
Wild Card characters * and ?	Two symbols, called wildcards, allow the user in DOS **to specify a groups of files.** The wild card character '*' indicates any group of characters while '?' indicates a single character	C:\> COPY *.doc C:\Temp\ C:\> DIR D:\Prog*.?

Appendix-V: UNIX Commands

Command	Use
date	Used to display the current system date and time
ls	Used to list directory contents
cp	Used to copy files
cat	Used to display the contents of files
mv	Used to rename files
rm	Used to delete files permanently
mkdir	Used to create a directory
rmdir	Used to remove a directory
cd	Used to change the current working directory
more	Used to display the contents of a file one page at a time
find	Used to search for a file within a given directory and its sub-directories
vi	Used to start the standard text editor
chmod	Used to change file permission

Appendix-VI: Index for C Commands, and Operations

Search Name	Page	Search Name	Page
#define directive	8-9	auto storage class	12-13
#include directive	8-4	break statement	11-14
++ and -- Operators as a statement	11-4	char type data	8-7
++ and -- Operators in an expression/condition	11-4	comments in C	8-4
actual arguments of a function	12-4	compound operators (+=, -=, *=, /=, %=)	11-1
address operator &	13-9	conditional operator (? :)	10-16
arithmetic and logical operations on a pointer	13-14	const keyword	8-10
arithmetic operators (+, -, *, /, %)	9-13	continue statement	11-15
array declaration	13-1	copying a structure / structure members	15-4
array of structures	15-5	decrement operator --	11-1
arrays and pointers	13-9	division operator and different data types	9-15

Search Name	Page	Search Name	Page
double type data	8-7	nested structure	15-2
do-while loop	11-16	nested while loops	11-12
entering and displaying strings	14-2	nesting of if-else statements	10-5
escape sequences (\n, \t, \b, \r, \\, \", \')	9-10	passing arrays to functions	13-8
extern storage class	12-15	passing structures to functions	15-8
field width specifiers:	9-8	pointer and strings	14-10
float type data	8-6	pointer declaration	13-10
for loop	11-18	precedence and associativity of operators	9-14
formal arguments of a function	12-5	printf( ) function	9-1
format specifiers (%d, %f, %c, %ld, %lf, %Lf)	9-1	putchar( ) function	9-11
function definition	12-3	puts( ) function	9-11
function prototype	12-2	register storage class	12-14
function prototype	12-6	relational operators (<, <=, >, >=, ==, !=)	10-1
function recursion	12-10	return statement	8-5
general rule for data type conversions	9-17	return statement	12-7
getchar( ) function	9-12	scanf( ) function	9-3
gets( ) function	9-13	short int type data	8-7
gets( ) function	14-3	static storage class	12-14
if without else statement	10-3	storage classes	12-12
if-else ladder	10-7	strcat( ) function	14-9
if-else	10-2	strcmp( ) function	14-7
increment operator (++)	11-1	strcpy( ) function	14-6
indirection operator (*)	13-10	strlen( ) function	14-5
initialising a structure	15-4	struct statement	15-1
initialising an array	13-4	structure containing arrays	15-7
int type data	8-6	switch-case-default statement	10-19
logical operators (AND &&, OR ||, NOT !)	10-10	typedef statement	15-3
long double type data	8-7	union type data	15-9
main() function	8-4	unions as members of other data types	15-11
matrix and strings	14-4	ways of declaring structure variables	15-2
matrix of structure type data	15-7	while loop	11-6
multi dimension arrays and matrices	13-5	while loop without using counter variable	11-10
nested for loops	11-20		

Some Amazing Facts About Computers !

- Majority of supercomputers today use different versions of the LINUX operating system
- A present day iPad2 has as much computing power as the Cray-2 supercomputer in 1985
- Prior to 1946, programs had to be wired into the computer, and the wirings needed to be changed to change a program
- On an average 12,000 laptops are lost or stolen in United States airports each week
- Early hard drives had 20MB capacity and cost about Rs.50,000/-, whereas a pen drive today holds 8GB memory and costs Rs.500/-
- In 1996, the European Space Agency's $1 billion Ariane 5 rocket had to be destroyed less than a minute after launch due to a bug in the on-board guidance program
- Bill Gates was ready to launch Windows under the name 'Interface Manager' before he was persuaded by an employee to change it
- You cannot create a folder and name it 'con' in Windows
- Macintosh was the first commercially successful personal computer that had a graphical user interface (GUI) and mouse instead of a command line interface

ANSWERS TO MCQ'S

Chapter 1

(i) b (ii) c (iii) d (iv) a (v) c (vi) b (vii) c (viii) a (ix) b (x) d
(xi) a (xii) c (xiii) d (xiv) a

Chapter 2

(i) b (ii) c (iii) a (iv) a (v) b (vi) d (vii) a (viii) b (ix) a (x) d
(xi) a (xii) c (xiii) d (xiv) d (xv) c (xvi) d (xvii) c (xviii) b (xix) c (xx) b
(xxi) a (xxii) d (xxiii) c (xxiv) a (xxv) a (xxvi) b (xxvii) a (xxviii) a (xxix) b (xxx) a
(xxxi) c (xxxii) b (xxxiii) a (xxxiv) c (xxxv) a (xxxvi) c (xxxvii) d (xxxviii) b (xxxix) b (xl) d
(xli) b (xlii) c (xliii) d (xliv) a (xlv) d (xlvi) b (xlvii) b (xlviii) c (xlix) c (l) b

Chapter 3

(i) c (ii) c (iii) d (iv) c (v) d (vi) b (vii) c (viii) d (ix) b (x) a
(xi) c (xii) c (xiii) b (xiv) c (xv) c (xvi) c (xvii) b (xviii) c (xix) b (xx) c
(xxi) b (xxii) a (xxiii) b (xxiv) b (xxv) c (xxvi) c (xxvii) c (xxviii) d (xxix) b (xxx) c
(xxxi) b (xxxii) c (xxxiii) b (xxxiv) c (xxxv) b (xxxvi) a (xxxvii) d (xxxviii) d (xxxix) c (xl) b
(xli) a (xlii) d (xliii) d (xliv) a (xlv) a (xlvi) a (xlvii) c (xlviii) b (xlix) b (l) d
(li) d (lii) b (liii) a (liv) d (lv) d (lvi) b (lvii) d

Chapter 4

(i) c (ii) a (iii) b (iv) c (v) d (vi) c (vii) d (viii) a (ix) c (x) d
(xi) a (xii) b (xiii) c (xiv) c (xv) b (xvi) b (xvii) c (xviii) d (xix) b (xx) a
(xxi) c (xxii) d (xxiii) a (xxiv) a (xxv) c (xxvi) c (xxvii) b (xxviii) d (xxix) a (xxx) b
(xxxi) b (xxxii) a (xxxiii) d (xxxiv) c (xxxv) b (xxxvi) a (xxxvii) d (xxxviii) a (xxxix) b (xl) a
(xli) d (xlii) c (xliii) b (xliv) c (xlv) d (xlvi) d (xlvii) a (xlviii) b (xlix) d (l) c
(li) c (lii) a (liii) d (liv) c (lv) a (lvi) a (lvii) d (lviii) d (lix) c (lx) b
(lxi) d (lxii) c (lxiii) b (lxiv) a (lxv) b (lxvi) b (lxvii) c (lxviii) b (lxix) c (lxx) d
(lxxi) c (lxxii) b (lxxiii) c (lxxiv) a (lxxv) b (lxxvi) c (lxxvii) a (lxxviii) b

Chapter 5

(i) b (ii) b (iii) c (iv) b (v) b (vi) c (vii) d (viii) b (ix) a (x) d
(xi) d (xii) c (xiii) d (xiv) a (xv) d (xvi) d (xvii) a (xviii) b (xix) b (xx) d
(xxi) c (xxii) b (xxiii) b (xxiv) c (xxv) d

Chapter 6

(i) a (ii) c (iii) d (iv) a (v) a (vi) b (vii) b (viii) c (ix) b (x) c
(xi) d (xii) d (xiii) b (xiv) b (xv) c (xvi) d (xvii) a (xviii) c (xix) c (xx) a
(xxi) b (xxii) c (xxiii) d (xxiv) c (xxv) b (xxvi) d (xxvii) a (xxviii) d (xxix) d (xxx) c
(xxxi) b (xxxii) d (xxxiii) c (xxxiv) a (xxxv) d (xxxvi) a (xxxvii) a (xxxviii) d (xxxix) b (xl) d
(xli) c (xlii) c (xliii) d (xliv) d (xlv) a (xlvi) b (xlvii) d (xlviii) a (xlix) c (l) a
(li) d (lii) b (liii) a (liv) a (lv) b (lvi) b (lvii) b (lviii) b (lix) a (lx) b
(lxi) a (lxii) c (lxiii) b (lxiv) c (lxv) d (lxvi) c (lxvii) b (lxviii) d (lxix) d (lxx) a
(lxxi) b (lxxii) c (lxxiii) d (lxxiv) d (lxxv) b (lxxvi) b (lxxvii) d (lxxviii) a (lxxix) d (lxxx) a

Chapter 7

(i) b (ii) c (iii) d (iv) b (v) c (vi) b

Chapter 8

(i) d (ii) d (iii) a (iv) c (v) c (vi) a (vii) c (viii) c (ix) d (x) d
(xi) a (xii) a (xiii) c (xiv) c (xv) d (xvi) b (xvii) d (xviii) c (xix) b (xx) b
(xxi) d

Chapter 9

(i) c (ii) a (iii) c (iv) d (v) c (vi) d (vii) b (viii) c (ix) d (x) b
(xi) a (xii) c (xiii) c (xiv) d (xv) b (xvi) c (xvii) a (xviii) c (xix) d (xx) a
(xxi) c (xxii) d (xxiii) b

Chapter 10

(i) b (ii) a (iii) c (iv) c (v) b (vi) d (vii) c (viii) d (ix) c (x) d
(xi) a (xii) c (xiii) b (xiv) d (xv) b (xvi) a (xvii) d (xviii) b (xix) d (xx) a
(xxi) a (xxii) b (xxiii) c (xxiv) b (xxv) c (xxvi) b (xxvii) d (xxviii) b (xxix) d (xxx) d

Chapter 11

(i) b (ii) b (iii) c (iv) c (v) c (vi) c (vii) a (viii) c (ix) d (x) d
(xi) c (xii) b (xiii) c (xiv) b (xv) c (xvi) c (xvii) d (xviii) a (xix) c (xx) c
(xxi) a (xxii) c (xxiii) c (xxiv) c (xxv) c (xxvi) d (xxvii) c (xxviii) b (xxix) c (xxx) a

Chapter 12

(i) c (ii) d (iii) b (iv) a (v) c (vi) d (vii) d (viii) a (ix) b (x) b
(xi) a (xii) d (xiii) b (xiv) c (xv) d (xvi) b (xvii) c (xviii) d (xix) c (xx) d
(xxi) b (xxii) a

Chapter 13

(i) c (ii) b (iii) d (iv) c (v) a (vi) b (vii) a (viii) c (ix) b (x) d
(xi) d (xii) b (xiii) c (xiv) b (xv) c (xvi) c (xvii) b (xviii) b (xix) b (xx) c

Chapter 14

(i) a (ii) b (iii) c (iv) d (v) b (vi) d (vii) c (viii) d (ix) b (x) c
(xi) a (xii) a (xiii) a (xiv) d (xv) c (xvi) b (xvii) d (xviii) a

Chapter 15

(i) b (ii) c (iii) a (iv) c (v) d (vi) c (vii) c (viii) d

QUESTIONS OF ANNUAL EXAMINATION, 2015
COMPUTER SCIENCE
CLASS - XI
(PRACTICAL)
(NEW SYLLABUS)

[Time : 3 hours] [Full Marks : 30]

Special credit will be given for answers which are brief and to the point.
Marks will be deducted for spelling mistakes, untidiness and bad handwriting.
Figures in the margin indicate full marks for the questions.

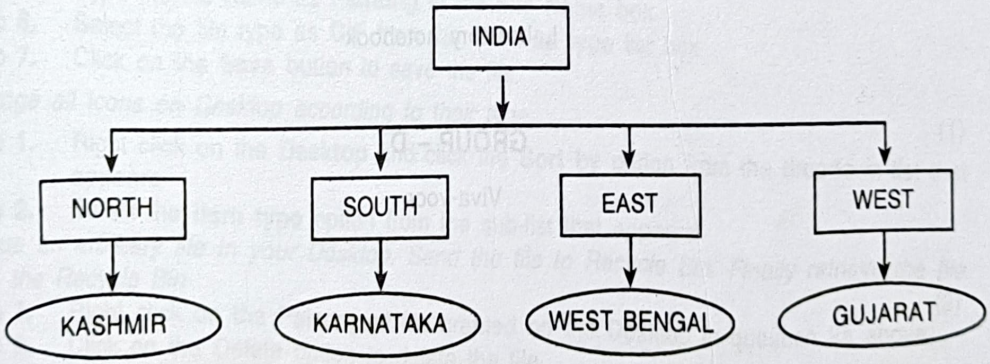
General Instructions :

- There are 4 (four) groups. Group-A contains 4 (four) questions. Group-B contains 2 (two) parts, each containing 4 (four) questions. You have to answer 1 (one) question from Group-A and 1 (one) question from each part of Group-B.
- Write all the steps in your answer-script which you have performed with the computer.
- Print all files, if necessary and possible, otherwise write all files with partial data input and output in your answer-script.
- Make suitable assumptions, if any, and tabulate them.

GROUP - A

Answer any *one* from the following questions : 5 x 1 = 5

1. Carry out the following using the Windows operating system :
   (a) Find all files with the file extension .rtf in your computer.
   (b) Create a folder on Desktop and Rename it as 'COMS PRACTICAL EXAM'.
   (c) Sort the files in C drive by name. 2 + 2 + 1
2. (a) Create a .bmp file using the MS-Paint program and then save it on the Desktop as .gif file.
   (b) Arrange all icons on Desktop according to their type.
   (c) Create an arbitrary file in your Desktop. Send the file to Recycle Bin. Finally retrieve the file from Recycle Bin. 2 + 1 + 2
3. Using Unix / Linux operating system commands, create the following tree : 5



Here boxes are denoting directories and circles are files.

4. (a) Using Unix/Linux command, display all current users of system.
   (b) Using Unix/Linux, create a file 'Test' containing the sentence "This is a line".
   (c) Make the file 'Test' hidden. Enable read and write permission of the file 'Test' for all users. 1 + 2 + 2

GROUP - B

PART - I

Answer any *one* from the following questions : 5 x 1 = 5

- Algorithm / Flowchart : 2

- Program coding in 'C' language : 2
- Execution and output : 1

1. Write a C program that checks whether an input alphabet is a vowel or consonant. (Check both lower case and upper case) 5
2. Write a program in C language to find out the summation of all odd numbers in between 1 to *n*, where *n* is a positive integer. After execution, show output. 5
3. Write a program in C language to print digits of an inputted number in reverse. 5
4. Write a C program to generate the following pattern : 5

```
1
1  3
1  3  5
1  3  5  7
1  3  5  7  9
```

PART – II

Answer any *one* from the following questions : $10 \times 1 = 10$

- Algorithm / Flowchart : 4
- Program coding in 'C' language : 4
- Execution and results : 2

1. Write a C program to find out the sum of *n* natural numbers using recursion, where *n* is a positive integer. 10
2. Write a C program to check whether an inputted string is Palindrome or not without using strrev() function. Display appropriate message as output. 10
3. Write a C language program to generate Armstrong Number.
   [A number is Armstrong if the sum of cubes of individual digits of a number is equal to the number itself, *e.g.*, $371 = 3^3 + 7^3 + 1^3$] 10
4. Write a program to generate a Transpose Matrix of a 3 x 3 matrix using C language. 10

GROUP – C

Laboratory notebook 5

GROUP – D

Viva-voce 5

COMPUTER SCIENCE PAPER SOLUTION
CLASS – XI
2015 Practical Examination
GROUP – A

Answer any one from the following :

1. ***Carry out the following using the Windows operating System:***

   (a) Find all files with file extension .rtf in your computer. (2)

   Step 1. Click on the Search button
   Step 2. Within the 'Search programs and files' box type the search condition **.rtf* and press Enter

   (b) Create a folder on Desktop and rename it as 'COMS PRACTICAL EXAM'. (2)

   Step 1. Right click on the Desktop
   Step 2. Click on the New option in the dropdown list that appears
   Step 3. Click on the Folder option from the dropdown sub-list that appears
   Step 4. A New Folder appears. Type the name of the folder as 'COMS PRACTICAL EXAM'

   (c) Sort the files in C drive by name. (1)

   Step 1. Double click on the **Computer** icon on the **Desktop**
   Step 2. Double click on the **C:** drive icon from under the **Hard Disk Drives** pane
   Step 3. The **C:** drive window opens
   Step 4. Right click on the white space in the right side pane
   Step 5. Select the **Sort** by option from the dropdown list that appears
   Step 6. Select the **Name** option from the sub-list that appears to sort the files by name

2. ***(a) Create a bmp file using MS-Paint program and then save it on the Desktop as a .gif file.*** (2)

   Step 1. Open Paint by clicking on **Start** → **All Programs** → **Accessories** → **Paint**
   Step 2. Draw something in **MS Paint**
   Step 3. Click on the **Save** icon at the left side of the **Title** bar
   Step 4. In the **Save As** window that opens, select the **Desktop** option from the left side pane
   Step 5. Type the file name as **Painting** in the **File Name** box
   Step 6. Select the file type as **GIF** from the **Save as type** list box
   Step 7. Click on the **Save** button to save the file

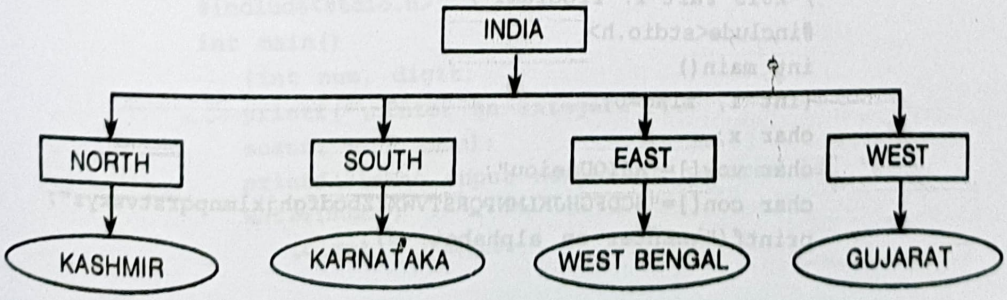
   (b) Arrange all icons on Desktop according to their type. (1)

   Step 1. Right click on the **Desktop** and click the **Sort by** option from the dropdown list that appears
   Step 2. Select the **Item type** option from the sub-list that appears

   (c) Create an arbitrary file in your Desktop. Send the file to Recycle Bin. Finally retrieve the file from the Recycle Bin (2)

   Step 1. Right click on the Painting.gif file created on the Desktop in question 2a above
   Step 2. Click on the Delete option to delete the file
   Step 3. Double click on the Recycle Bin icon on the Desktop
   Step 4. Locate the deleted file in the Recycle Bin window that opens and right click on it
   Step 5. Select the Restore option from the dropdown list that appears to retrieve the file

3. ***Using UNIX / Linux operating system commands create the following tree:*** (5)



Note: Here boxes denoting directories and circles are files.

Step 1. Type the following in the UNIX prompt to create the directory INDIA:
mkdir INDIA

Step 2. Move into the INDIA directory by typing the following:
cd /INDIA

Step 3. Type the following in the UNIX prompt to create the directories NORTH, SOUTH, EAST, WEST under the INDIA directory
mkdir NORTH SOUTH EAST WEST

Step 4. The cat command can be used to create the following files under the directories:
cat > /INDIA/NORTH/KASHMIR.TEXT

Step 5. Type any text in the file and press Enter followed by *Ctrl+D* to save and close the file

Step 6. Next type the following:
cat > /INDIA/SOUTH/KARNATAKA.TEXT

Step 7. Type any text in the file and press Enter followed by *Ctrl+D* to save and close the file

Step 8. Next type the following:
cat > /INDIA/EAST/WEST_BENGAL.TEXT

Step 9. Type any text in the file and press Enter followed by *Ctrl+D* to save and close the file

Step 10. The cat command can be used to create the files under the directories as shown above:
cat > /INDIA/WEST/GUJARAT.TEXT

Step 11. Type any text in the file and press Enter followed by *Ctrl+D* to save and close the file

4. (a) *Using Unix/Linux command display all current users of the system.* (1)

Step 1. Type the following command and press Enter to display the current users:
who

(b) Using Unix/Linux create a file 'Test' containing the sentence "This is a line". (2)

Step 1. Open the vi editor in UNIX to create the file 'Test' by typing the following:
vi Test

Step 2. Type i and press **Enter**

Step 3. Type the sentence "*This is a line*" and press **Enter** followed by **Esc** to get back to the command mode

Step 4. To save the file, type :*wq!* and press **Enter** to save and exit the vi editor

(c) Make the file 'Test' hidden. Enable read and write permission of the file 'Test' for all users.(2)

Step 1. Type the following under the command prompt to make the file 'Test' hidden (put a dot i.e. (.) in front of the file name by renaming the file):
mv Test .Text

Step 2. Type the following under the command prompt to enable read and write permission of the file 'Test' for all users:
chmod 006 Test

GROUP – B

Part – 1

Answer any one from the following : 5x1=5

1. *Write a C program that checks whether an input alphabet is a vowel or a consonant. (Check both lowercase and uppercase):*

Step 1. Open the C programming IDLE (like Turbo C or DevC++) and type the following code in a new file:

```
/*2015 Part-1: Program1*/
#include<stdio.h>
int main()
{int i, flag=0;
char x;
char vow[]="AEIOUaeiou";
char con[]="BCDFGHJKLMNPQRSTVWXYZbcdfghjklmnpqrstvwxyz";
printf("\nEnter an alphabet: ");
```

```
fflush(stdin);
scanf("%c", &x);
for(i=0; vow[i]!='\0'; i++)
   {if(x==vow[i])
      {printf("\nVowel");
      flag==1;
      break;
      }
   }
for(i=0; flag!=1 && con[i]!='\0'; i++)
   {if(x==con[i])
      {printf("\nConsonent");
      flag==1;
      break;
      }
   }
if(flag==0)
   printf("\nAny other character");
fflush(stdin);
getchar();
return 0;
}
```

Step 2. Save the file as part1_P1.C and compile it.

Step 3. Run the program after compilation.

2. ***Write a program in C language to find out the summation of all odd numbers in between 1 and n, where n is a positive integer. After execution show output.***

Step 1. Open the C programming IDLE and type the following code in a new file:

```
/*2015 Part-1: Program2*/
#include<stdio.h>
int main()
{int i, n, sum=0;
printf("\nEnter an integer: ");
scanf("%d", &n);
for(i=1; i<=n; i+=2)
   sum = sum + i;
printf("\nRequired sum = %d", sum);
fflush(stdin);
getchar();
return 0;
}
```

Step 2. Save the file as part1_P2.C and compile it.

Step 3. Run the program after compilation.

3. ***Write a program in C language to print digits of an input number in reverse.***

Step 1. Open the C programming IDLE and type the following code in a new file:

```
/*2015 Part-1: Program3*/
#include<stdio.h>
int main()
   {int num, digit;
   printf("\nEnter an integer: ");
   scanf("%d", &num);
   printf("\nThe input number in reverse is: ");
   while(num>0)
      {digit = num%10;
```

```
        printf("%d", digit);
        num = num/10;
        }
    fflush(stdin);
    getchar();
    return 0;
    }
```

Step 2. Save the file as part1_P3.C and compile it.

Step 3. Run the program after compilation.

4. ***Write a C program to generate the following pattern:***

```
1
1 3
1 3 5
1 3 5 7
1 3 5 7 9
```

Step 1. Open the C programming IDLE and type the following code in a new file:

```
/*2015 Part-1: Program4*/
#include<stdio.h>
int main()
   {int i, j;
   for(i=1; i<=5; i++)
      {for(j=1; j<=i; j++)
            printf("%d ", 2*j-1);
      printf("\n");
      }
   fflush(stdin);
   getchar();
   return 0;
   }
```

Step 2. Save the file as part1_P4.C and compile it.

Step 3. Run the program after compilation.

Part-II

Answer any one from the following: 10 x 1 = 10

1. ***Write a C program to find out the sum of n natural numbers using recursion, where n is a positive integer.***

Step 1. Open the C programming IDLE and type the following code in a new file:

```
/*2015 Part-2: Program1*/
#include<stdio.h>
int sum(int n)
   {int sum;
   if(n==0)
      return 0;
   sum = n + sum(n-1);
   return sum;
   }
int main()
   {int n, s;
   printf("\nEnter an integer: ");
   scanf("%d", &n);
   s = sum(n);
   printf("\nRequired sum = %d", s);
   fflush(stdin);
```

```
getchar();
return 0;
}
```

Step 2. Save the file as part2_P1.C and compile it.

Step 3. Run the program after compilation.

2. ***Write a C program to check whether an input string is Palindrome or not without using strrev() function. Display appropriate message as output:***

Step 1. Open the C programming IDLE and type the following code in a new file:

```
/*2015 Part-2: Program2*/
#include<stdio.h>
#include<string.h>
int main()
   {int i, len, flag=1;
   char word[80];
   printf("\nEnter a word: ");
   gets(word);
   len = strlen(word);
   for(i=0; i<len/2; i++)
      { if(word[i] != word[len-i-1])
          {flag=0;
          break;
          }
      }
   if(flag==1)
      printf("\nPalindrome");
   else
      printf("\nNot a Palindrome");
   fflush(stdin);
   getchar();
   return 0;
   }
```

Step 2. Save the file as part2_P2.C and compile it.

Step 3. Run the program after compilation.

3. ***Write a C program to generate Armstrong number. [A number is an Armstrong number if the sum of the cubes of individual digits of a number is equal to the number itself, e.g. 371 = 3^3 + 7^3 + 1^3]***

Step 1. Open the C programming IDLE and type the following code in a new file:

```
/*2015 Part-2: Program3*/
#include<stdio.h>
int main()
   {int num, copy, digit, sum;
   for(num=100; num<=999; num++)
      {sum = 0;
      copy = num;
      while(copy>0)
          {digit = copy%10;
          copy = copy/10;
          sum = sum + digit*digit*digit;
          }
      if(sum == copy)
          printf("\n%d", num);
      }
   fflush(stdin);
```

```
    getchar();
    return 0;
    }
```

Step 2. Save the file as part2_P3.C and compile it.

Step 3. Run the program after compilation.

4. ***Write a program to generate a Transpose Matrix of a 3x3 matrix using C language.***

Step 1. Open the C programming IDLE and type the following code in a new file:

```
/*2015 Part-2: Program4*/
#include<stdio.h>
int main()
   {int mat[3][3], trans[3][3], i, j;
   for(i=0; i<3; i++)
      {for(j=0; j<3; j++)
         {printf("\nEnter value[%d][%d] ", i, j)
         scanf("%d", &mat[i][j]);
         trans[j][i] = mat[i][j];
         }
   printf("\nThe Transpose Matrix...\n");
   for(i=0; i<3; i++)
      {for(j=0; j<3; j++)
         printf("\t%d", trans[i][j])
      printf("\n");
      }
   fflush(stdin);
   getchar();
   return 0;
   }
```

Step 2. Save the file as part2_P4.C and compile it.

Step 3. Run the program after compilation.

QUESTIONS OF ANNUAL EXAMINATION, 2015
COMPUTER SCIENCE
CLASS – XI
New Syllabus

Total Time : 3 Hours 15 minutes] [Full Marks : 70

Special credit will be given for answers which are brief and to the point.
Marks will be deducted for spelling mistakes, untidiness and bad handwriting.
Figures in the margin indicate full marks for the questions.

GROUP – A

1. Answer the following questions (MCQ type) : $1 \times 21 = 21$

(i) Which of the following is not an Input Device ?
(a) Scanner (b) CRT Monitor (c) Touch Screen (d) Barcode Reader.

(ii) U.L.S.I. stands for –
(a) Ultra Large Scale Integration (b) Under Large Scale Integration
(c) Ultra Large Scale Integer (d) Ultraviolet Low Scale Integer.

(iii) Main component of 2nd generation computer is –
(a) Vaccum Tubes (b) Transistors
(c) Integrated Circuits (d) Mega Chips.

(iv) Cache memory is situated in between –
(a) Primary and Secondary Memory (b) Register and Secondary Memory
(c) Register and Primary Memory (d) Primary Memory and Flash Drive.

(v) Which one of the following is not an example of 5th generation computing ?
(a) Artificial Intelligence (b) Voice Recognition
(c) Parallel Processing (d) Integrated Circuits.

(vi) $A + A + O + \overline{A} = ?$
(a) 1 (b) 0 (c) A (d) $\overline{A}$

(vii) In SOP expression $AB + \overline{B}C$ of truth table, how many of the outputs are low ?
(a) 1 (b) 2 (c) 4 (d) 5.

(viii) NOT GATE is also known as –
(a) Convertor (b) Decoder (c) Invertor (d) Universal Gate.

(ix) How many selection lines are required in 1 × 16 Demultiplexer ?
(a) 3 (b) 8 (c) 4 (d) 16.

(x) The terms of SOP expressions are called –
(a) Logical Terms (b) Max Terms (c) Boolean Terms (d) Min Terms

(xi) $(4)_g + (6)_g = (?)_g$
(a) 10 (b) 11 (c) 12 (d) 13.

(xii) _____ stores the address of another variable.
(a) Structure (b) Array (c) Union (d) Pointer.

(xiii) Which of the following commands is used to copy in UNIX operating system ?
(a) CD (b) CP (c) PC (d) DC

(xiv) Which of the following is used as a wildcard character in DOS ?
(a) & (b) # (c) ! (d) ?

(xv) _____ is used to take user input through keyboard in C language.

(a) printf (b) scanf (c) print (d) scan.

(xvi) int i :

for (i = 0; i < = 4;)

printf .....

Correct output is –

(a) 1 2 3 4 (b) 0 1 2 3 (c) 1 2 3 4 5 (d) 1 2 3.

(xvii) Which of the following is not an Application Software ?

(a) Foxpro (b) Assembler (c) Games (d) Spreadsheet.

(xviii) Which is used to represent 'NULL' character in 'C' Language ?

(a) \n (b) \t (c) \o (d) \u.

(xix) Which of the following is a Logical Operator in 'C' Language ?

(a) > (b) < (c) ! = (d) !.

(xx) Which of these is a correct pointer declaration in C ?

(a) int *ptr (b) *int ptr (c) int ptr* (d) int &ptr.

(xxi) Which header file includes gets ( ) and puts ( ) function ?

(a) string.h (b) conio.h (c) stdio.h (d) math.h

GROUP – B

2. Answer the following questions in brief (Alternatives are to be noted) : $1 \times 14 = 14$

(i) Which library function is used to convert a given string to uppercase ?

Or, Which library function is used to copy a string ?

(ii) In which generation of computers Microprocessor is used ?

Or, Name the components used in first generation computers.

(iii) Write down the Truth Table of 2-Input NAND gate.

Or, Write down the Truth Table of a Half Adder.

(iv) What does VLSI mean ?

Or, What is EEPROM ?

(v) What is time sharing operating system ?

(vi) Give two examples of conditional operators in 'C'.

Or, Show an application of binary operator in 'C'.

(vii) What is Recursion ?

(viii) What is the purpose of Header files in 'C' ?

Or, What are storage classes in 'C' ?

(ix) Differentiate between 'While-do' and 'Do-while' statements.

Or, Differentiate between 'break' and 'exit' statements in 'C'.

(x) What do you mean by a Switch case statement ?

(xi) What will be the output of the following ?

printf ("% 5.3f"; 125.1)k

Or, Which library function is used for counting number of characters in a string ?

(xii) What is infinite loop in C ?

Or, What is function prototype ?

(xiii) Let *a* and *b* have values 20 and 30 respectively. Write a C code to interchange the values of these two variable without using a third variable.

(xiv) Represent $(-6)_{10}$ in 4-bit 1's complement form.

Or, What is 2's complement representation ?

GROUP – C

3. Answer the following questions : (Alternatives are to be noted) $7 \times 5 = 35$

(i) (a) Find $(x)_{10} + (11001)_2 = (57)_8$

(b) $(61)_8 + (45)_8 = (?)_8$

(c) $(A\ 6\ D)_{16} = (?)_8$

Or, (a) $(1011.10)_8 = (?)_{10}$

(b) $(BCE)_{16} + (17D)_{16} = (?)_{16}$

(c) $(285)_{10} = (?)_{16}$

(ii) (a) Differentiate between 'Call by value' and 'Call by reference' using suitable examples.

(b) Write a C program to calculate the 'Factorial' of any inputted number of given run-time.

Or, (a) What is an array? Declare an array to store 20 real numbers.

(b) Write a program in C to add two 3×3 matrices.

(iii) (a) Draw the block diagram of 4×1 Multiplexer and the corresponding Truth Table.

(b) Find out the logic expression of $f(A, B, C) = \Sigma(1, 2, 4) + d(0, 7)$.

Or, (a) Draw the diagram of octal to binary encoder and the corresponding Truth Table.

(b) What is a 'Full Subtractor' ?

(iv) (a) State the functions of operating system.

(b) Write two differences between internal and external commands.

(c) What is Utility Software ? Give Example. 2

(v) (a) What is 'Spooling'? 2

(b) What is the function of L.S. command in Unix ? 2

(c) What is a 'Loader' ? 3

Or, (a) Write a program in C to make the following structure 4
(The Number of lines will be taken as Input).

```
*
* *
* * *
* * * *
* * * * *
```

(b) Write two utilities of flowchart. Mention the different components of a flowchart. 3

Or, (a) Write a program in C to print the following series (1, 5, 9, ...n), where n is the upper limit. 5

(b) What is the purpose of STRCMP ( ) function ? 2

COMPUTER SCIENCE PAPER SOLUTION
CLASS – XI
2015 Annual Examination

GROUP – A

1. Answer the following questions (MCQ type) :

(i) (b) CRT Monitor
(ii) (a) Ultra Large Scale Integration
(iii) (b) Transistor
(iv) (c) Register and Primary Memory
(v) (d) Integrated Circuits
(vi) (a) 1
(vii) (c) 4
(viii) (c) Invertor
(ix) (c) 4
(x) (d) Min terms
(xi) (c) 12
(xii) (d) Pointer
(xiii) (b) CP
(xiv) (d) ?
(xv) (b) scanf
(xvi) (c) 1 2 3 4 5
(xvii) (b) Assembler
(xviii) (c) \0
(xix) (d) !
(xx) (a) int *ptr
(xxi) (c) stdio.h

GROUP – B

2. Answer the following questions in brief (Alternatives are to be noted) :

(i) toupper ( )

OR

strcpy ( )

(ii) 4th generation

OR

Vacuum Tube

(iii) Truth table of 2 input NAND gate

A	B	$\overline{A.B}$
0	0	1
0	1	1
1	0	1
1	1	0

OR

Truth table of Half Adder

x	y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(iv) VLSI stands for Very Large Scale Integration. It is a technology used to make integrated circuits.

OR

EEPROM stands for Electrically Erasable Programmable Read Only Memory. It is a type of ROM.

(v) In a Time Sharing Operating System the total time of the OS is shared among multiple users.

(vi) Two examples of conditional operators are :

```
max = (a > b) ? (a) : (b) ;
amount = (cost > 5000) ? (cost * 0.9) : (cost * 0.95);
```

OR

Example of binary operator :

```
x = y + z ;
```

(vii) Recursion is the process where a function calls itself. Factorial of a number can be calculated using a recursive function for example.

(viii) Header files are used in C to provide prototypes of library functions used in the program.

OR

The storage class of a variable in C determines the scope and class of a variable *i.e.*, it determines in which location of the computer the values of the variables are stored.

(ix) A **while** loop is an entry controlled loop where the condition is checked at the beginning of the loop, whereas a **do-while** loop is an exit controlled loop where the condition is checked at the end of the loop.

OR

A **break** statement takes the control out of a loop *i.e.*, terminates a loop. An **exit** statement terminates a program.

(x) A switch case statement is an alternative to if-else statement and can be used to test a value and execute certain statements based on the value.

(xi) Output : 125.1

OR

The strlen ( ) library function is used to count the number of characters in a string.

(xii) An infinite loop never ends and goes on forever.

e.g., for (; 1 > 0 ;) ;

OR

A function prototype is a declaration of a function that specifies the function's name and signature. It includes the function return data type, the function name, the function parameter list.

(xiii) Code to exchange the values of two variables without using third variable.

```
int a = 6, b = 8 ;
a = a + b ;
b = a – b ;
a = a – b ;
```

(xiv) 4 bit 1's complement form of $(-6)_{10}$:

2 | 6	rem 0
2 | 3	1
2 | 1	1
0	

$\therefore 6_{10} = 110_2$

Positive 4 bit sign magnitude form of +6 is :

0	1	1	0

$\therefore$ Negative 1's complement form of – 6 is :

1	0	0	1

OR

Two's complement representation is used to represent negative integer numbers. To form the 2's complement :

1. Get the positive sign magnitude representation of the number

2. Get the 1's complement
3. Add 1 to the LSB of the 1's complement.

GROUP – C

3. Answer the following questions : (Alternatives are to be noted)

(i) (a) $(x)_{10} + (11001)_2 = (57)_8$

or, $(x)_{10} = (57)_8 - 31_8$

or, $(x)_{10} = (26)_8$

$\therefore \quad x = 2 * 8^1 + 6 * 8^0$

$= 16 + 6$

$= 22_{10}$ Ans

R.W.

$\underbrace{011}_{(3}\ \underbrace{001}_{1)_8}{}_2$

$$\begin{array}{r} 57 \\ -31 \\ \hline 26 \end{array}$$

(b) $(61)_8 + (45)_8 = (?)_8$

Ans. $(126)_8$

R.W.

$$\begin{array}{r} 61 \\ +45 \\ \hline 126_8 \end{array}$$

Alternative

$$\begin{array}{r} {\scriptstyle 1}\ 61 \\ +45 \\ \hline 106 \\ -8 \\ \hline 2 \end{array}$$

$= 126$

(c) $(A\ 6\ D)_8 = (?)_8$

A 6 D

$\underbrace{101}_{5}\ \underbrace{001}_{1}\ \underbrace{101}_{5}\ \underbrace{101}_{5}$

Ans : $(5155)_8$

OR

a) $(1011.10)_2 = (\ ?)_{10}$

$= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2}$

$= 8 + 0 + 2 + 1 + \frac{1}{2}$

$= 11 + \frac{1}{2}$

$= \frac{23}{2} = 11.5$ Ans.

(b) $(BCE)_{16} + (17D)_{16} = (?)_{16}$

$$\begin{array}{r} B\ C\ E \\ +\ 1\ 7\ D \\ \hline D\ 4\ B \end{array} \equiv \begin{array}{rrr} {\scriptstyle 1} & {\scriptstyle 1} & \\ 11 & 12 & 14 \\ +\ 1 & 7 & 13 \\ \hline 13 & 20 & 27 \\ & -16 & -16 \\ \hline 13 & 4 & 11 \\ \downarrow & \downarrow & \downarrow \\ D & 4 & B \end{array}$$

(c) $(285)_{10} = (?)_{16} = (11D)_{16}$

$$\begin{array}{r|l} 16 & 285 \\ 16 & 17 \quad \text{rem } 13 \uparrow \ D \\ 16 & 1 \qquad\qquad\ \ 1 \quad\ 1 \\ & 0 \qquad\qquad\ \ 1 \quad\ 1 \end{array}$$

(ii) Call by value occurs when a function is called with the value of a variable, whereas call by reference occurs when a function is called with the address of a variable. In call by value, if any

modification is done on the formal argument then that modification does not affect the actual argument. In call by reference, any modification done on the formal argument changes the actual argument.

Example :

```
# include <stdio.h>
void squareV ( int x )
      { x = x * x;
      printf ("%d", x);
      }
void squareR (int * x)
      { * x = (* x) * (* x);
      printf (" %d", * x);
      }
void main ( )
      { int y;
      scanf (" %d", &y);          // Let value input = 6
      squareV (y);                // Call by value of function squareV
      printf ("%d", y);           // Value of y printed will be 6
      squareR (&y);               // Call by reference of function square R
      printf ("%d", y);           // Value of y printed will be 36
      }
```

b)

```
#include <stdio.h>
/* To find factorial of a number */
void main ( )
{  int num, i ;
   long int fact = 1;
   printf ("Enter number :"); scanf ("%d", &num);
   for (i = 1; i < = num; i++)
       fact = fact * i ;
   printf ("\n Required factorial is % ld", fact);
}
```

(ii) OR

An array is an inbuilt linear data structure that can be used to store a set of values of the same data type. Each value stored in the array can be accessed using an index along with the array name. All values in the array occupy consecutive memory locations.

An array to store 20 real numbers is declared as :

```
float arr [20];
```

(b)

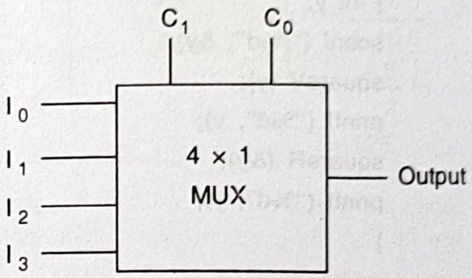
```
/* Program to Add two 3 x 3 matrices */
#include <stdio.h>
void main ( )
{  int A [3] [3], B [3] [3], S [3] [3], i, j ;
   for (i = 0; i < 3; i+ +)
       { for (j = 0; j < 3; j+ +)
           { printf ("\nEnter value of A :");
             scanf ("%d", &A [i] [j]);
```

```
                printf ("\n Enter value of B :");
                scanf ("%d", &B [i] [j]);
              }
        }
    for {i = 0; i < 3; i++)
      { for (j = 0; j < 3; j++)
          {S [i] [j] = A [i] [j] + B [i] [j];
           printf ("%d\t", S [i] [j]);
          }
        printf ("\n");
      }
    }
```

(iii) Truth Table of a 4 × 1 multiplexer :

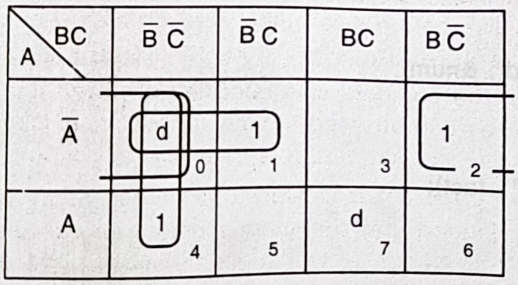
(a)

C_1	C_0	Output
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3



Block Diagram of a 4 × 1 MUX

(b) $f(A, B, C) = \Sigma (1, 2, 4) + d (0, 7)$



Term 1 = $\bar{B}\,\bar{C}$

Term 2 = $\bar{A}\,\bar{B}$

Term 3 = $\bar{A}\,\bar{C}$

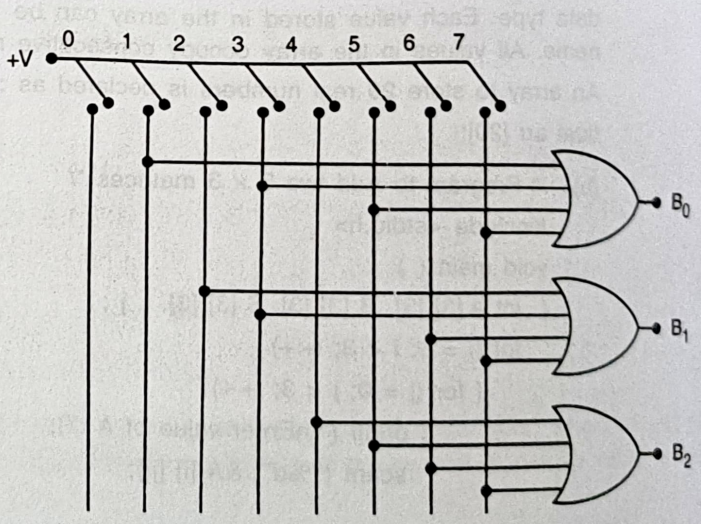
$\therefore f = \bar{A}\,\bar{B} + \bar{B}\,\bar{C} + \bar{A}\,\bar{C}$

(iii) OR

(a) Octal to binary encoder :

Truth Table

Octal	B_2	B_1	B_0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1



(b) A full subtractor circuit is used to subtract two binary digits. It has three inputs and two outputs.
The inputs are : Minuend (A), subtrahend (B), Borrow (B_{in})
The outputs are : Difference (D), Borrow out (B_0)

(iv) (a) The functions of operating system are :

(1) File Management : The file management module of the OS deals with creating, naming, storing, retrieving, organising and security of files.

(2) Memory Management : This module is responsible for the allocation and deallocation of memory space to various programs.

(3) Process Management : This module is responsible for the creation, deletion and running of several processes simultaneously.

(4) Device Management : This module keeps track of the input/output requests from various processes and issues commands to the input/output devices.

(5) Command Interpretation : This module takes care of interpreting the user commands.

(b) Internal Command	External Command
(i) These commands are automatically loaded at the time of booting of DOS.	(i) These commands are kept in the hard disk until they are needed.
(ii) These commands programs are not displayed at the time of listing the files	(ii) These command programs are displayed at the time of listing the files
(iii) *E.g.* DIR, CD, MD	(iii) *E.g.* FORMAT, MOVE, ATTRIB

(c) Utility programs are a common set of library programs that are used to do certain utility jobs like finding files, compressing files, defragment disks etc.

(iv) OR

(a) Spooling stands for simultaneous peripheral operations online. It is a technique used to solve the problem of speed mismatch between the processor and peripheral devices like printers, keyboards etc. Spooling places all the data that comes from a slow input device on the hard disk before it is loaded into the main memory. Similarly after execution by the CPU data in the main memory is written to the hard disk before it is output through a slow output device like a printer.

(b) The LS command of UNIX is used to list the contents of a specified directory. It is similar to the DIR command of DOS. Several switches can be used to view file listings in different forms like listing all contents including hidden files, or list contents of a directory in long form etc.

(c) The executable program code produced by the linker is stored in the secondary storage. To run the code the program needs to be loaded into the main memory for execution by the operating system. This job is done by the loader. It works along with a program called a relocator that helps to adjust the relative addresses of the different sub routines in the executable program as per the requirement of the primary memory.

(v) (a)
```
/* Program to Print Pattern */
# include < stdio.h >
void main ( )
  {  int i, n, j;
     printf ("\n Enter number of lines to print : ");
     scanf ("%d", & n);
     for (i = 1; i < = n; i++)
         { for (j = 0; j <= i; j++)
             printf (" * ");
           printf ("\n");
         }
  }
```

(b) Two utilities of a flowchart are :

(i) A flowchart gives a pictorial/graphical representation of the program logic.

(ii) The same flowchart can be used to develop a program using any programming language.

(v) OR

(a)
```
/* Program to print the series 1, 5, 9, ...., n * /
# include < stdio.h >
void main ( )
  {  int term, n ;
     printf ("\n Enter the term value up to which to print : ");
     scanf ("%d", & n);
     term = 1;
     while (term < = n)
         { printf ("\n %d", term);
           term = term + 4;
         }
  }
```

(b) The strcmp function is used to compare two strings. If the strings are str 1 and str 2, then strcmp returns 0 if both the strings are same. It returns –1 if the strings are in alphabetic order. It returns +1 if the strings are not in alphabetic order.

QUESTIONS OF ANNUAL EXAMINATION, 2016
COMPUTER SCIENCE
CLASS – XI
(PRACTICAL)
(NEW SYLLABUS)

[Time : 3 hours] [Full Marks : 30]

Special credit will be given for answers which are brief and to the point.
Marks will be deducted for spelling mistakes, untidiness and bad handwriting.
Figures in the margin indicate full marks for the questions.

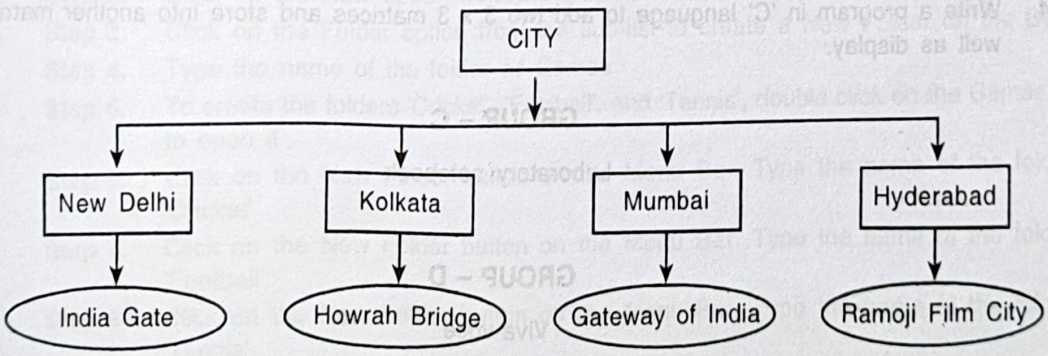
General Instructions :

- There are 4 (four) groups. Group-A contains 4 (four) questions. Group-B contains 2 (two) parts, each containing 4 (four) questions. You have to answer 1 (one) question from Group-A and 1 (one) question from each part of Group-B.
- Write all the steps in your answer-script which you have performed with the computer.
- Print all files, if necessary and possible, otherwise write all files with partial data input and output in your answer-script.
- Make suitable assumptions, if any, and tabulate them.

GROUP – A

Answer any *one* from the following questions : 5 x 1 = 5

1. Carry out the following using the Windows operating system :
   (a) Rename any one existing folder of the Desktop.
   (b) Convert all files and folders into 'Tiles' view of any one drive.
   (c) Change the Desktop Background.
   (d) Find out the files having '.txt' file extension name of any drive. 1 + 1 + 1 + 2
2. (a) Create a folder 'Games' on the Desktop. Inside this folder create three more folders namely 'Cricket', 'Football' and 'Tennis'.
   (b) Delete the 'Football' folder and again retrieve the folder at its original position. 3 + 2
3. Using Unix / Linux operating system commands, create the following tree : 5



Here boxes are denoting directories and oval-shapes are files.

4. (a) Show the list of all current users using Unix / Linux command.
   (b) Create a file named 'INDIA' which will contain "I love my country" using suitable command in Unix / Linux.
   (c) By using Unix / Linux command, show the present working directory and system date. 1 + 2 + 2

GROUP – B

PART – I

Answer any *one* from the following questions : $5 \times 1 = 5$

- Algorithm / Flow chart : 2
- Program coding in 'C' language : 2
- Execution and output : 1

1. Write a program in 'C' language to check an integer number whether it it even or odd. The number should be given as input except zero. 5
2. Write a 'C' program to take two strings as input and concatenate them. 5
3. Write a program in 'C' language to check an integer number whether it is prime or not. The number will be taken as input. 5
4. Write a program in 'C' language to make the following pattern :

```
* * * *
* * *
* *
*
```

5

PART – II

Answer any *one* from the following questions : $10 \times 1 = 10$

- Algorithm / Flow chart : 4
- Program coding in 'C' language : 4
- Execution and results : 2

1. Write a program in 'C' language, which will take an array of 5 numbers and find out the largest and smallest numbers of that array. 10
2. Write a 'C' program which can print the following series :

   0, 1, 1, 2, 3, 5, ..., *n* (Fibonacci series)

   Value of *n* will be inputted from user. 10
3. Write a program in 'C' language to convert all the lower case letters in a sentence to the corresponding upper case letters. 10
4. Write a program in 'C' language to add two 3 x 3 matrices and store into another matrix and as well as display. 10

GROUP – C

Laboratory notebook 5

GROUP – D

Viva-voce 5

COMPUTER SCIENCE PAPER SOLUTION
CLASS – XI
2016 Practical Examination

GROUP – A (Multiple Choice Questions)

Answer any one from the following : 5x1=5

1. ***Carry out the following using the Windows operating System:***

(a) Rename any one existing folder of the Desktop (1)

Step 1. Right click on a folder on the Desktop
Step 2. From the dropdown list select the Rename option.
Step 3. Type the new name of the folder

(b) Convert all files and folders into 'Tiles' view of any one drive. (1)

Step 1. Double click on the Computer icon on the Desktop
Step 2. Double click on the drive icon to open the drive to change view
Step 3. Click on More Options button on the right side of Menu bar for different view options
Step 4. Move the slider to get the 'Tiles' view

(c) Change the Desktop background. (1)

Step 1. Right click on the **Desktop**
Step 2. Click on the Personalize option from the drop-down list
Step 3. Click on the Desktop Background option
Step 4. Select a picture
Step 5. Click on Save Changes button to apply the Desktop Background

(d) Find out the files having '.txt' file extension name of any drive. (2)

Step 1. Click on the Start button at the left of the Task Bar
Step 2. Type the search criteria *.txt in the Search Box
Step 3. Press Enter to view files with this extension

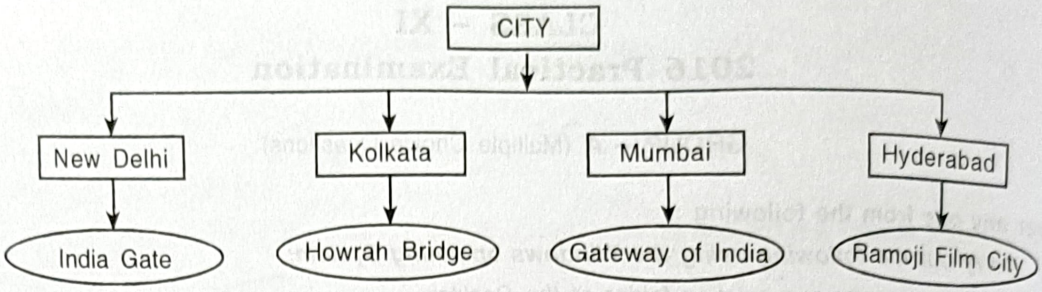
2. ***(a) Create a folder 'Games' on the Desktop. Inside this folder create three more folders namely 'Cricket', 'Football', and 'Tennis'.*** (2)

Step 1. Right click on the Desktop
Step 2. Click on the New option from the drop-down list
Step 3. Click on the Folder option from the sub-list to create a New Folder on the Desktop
Step 4. Type the name of the folder as Games
Step 5. To create the folders 'Cricket', 'Football', and 'Tennis', double click on the Games folder to open it
Step 6. Click on the New Folder button on the Menu Bar. Type the name of the folder as 'Cricket'
Step 7. Click on the New Folder button on the Menu Bar. Type the name of the folder as 'Football'
Step 8. Click on the New Folder button on the Menu Bar. Type the name of the folder as 'Tennis'

(b) Delete the 'Football' folder and again retrieve the folder at its original position. (2)

Step 1. Double click on the Games folder on the Desktop to open it
Step 2. Right click on the 'Football' folder and click on the Delete option from the drop-down list
Step 3. Click on OK to confirm the deletion
Step 4. Double click on the Recycle Bin icon on the Desktop
Step 5. Locate the deleted folder in the Recycle Bin window that opens, and right click on it
Step 6. To retrieve the folder, click on the Restore option from the dropdown list

3. ***Using UNIX / Linux operating system commands create the following tree:*** (5)



Note: Here boxes denoting directories and oval-shapes are files.

Step 1. Type the following in the UNIX prompt to create the directory CITY:
mkdir CITY

Step 2. Move into the CITY directory by typing the following:
cd /CITY

Step 3. Type the following in the UNIX prompt to create the directories 'New Delhi', 'Kolkata', 'Mumbai', 'Hyderabad' under the CITY directory
mkdir NewDelhi Kolkata Mumbai Hyderabad

Step 4. The cat command can be used to create the following files under the directories:
cat > /CITY/NewDelhi/IndiaGate.txt

Step 5. Press Enter. Type any text in the file and press Enter followed by Ctrl+D to save and close the file

Step 6. Next type the following:
cat > /CITY/Kolkata/HowrahBridge.txt

Step 7. Press Enter. Type any text in the file and press Enter followed by Ctrl+D to save and close the file

Step 8. Next type the following:
cat > /CITY/Mumbai/GatewayOfIndia.txt

Step 9. Press Enter. Type any text in the file and press Enter followed by Ctrl+D to save and close the file

Step 10. The cat command can be used to create the following files under the directories:
cat > /CITY/Hederabad/RamojiFilmCity.txt

Step 11. Press Enter. Type any text in the file and press Enter followed by Ctrl+D to save and close the file.

4. (a) **Show the list of all current users using Unix/Linux command.** (1)

Step 1. Type the following command and press Enter to display the current users:
who

(b) *Create a file named 'INDIA' which will contain "I love my country" using suitable command in Unix/Linux.* (2)

Step 1. The cat command can be used to create the file as shown below. Type the following under the Unix prompt:
cat > INDIA.txt

Step 2. Press Enter and type the following text: "I love my country"

Step 3. Press Enter followed by Ctrl+D to save and close the file

(c) *By using Unix/Linux command, show the present working directory and system date.* (2)

Step 1. To display the present working directory type the following command in the Unix prompt:
pwd

Step 2. Type the following under the command prompt to display the system date:
date

GROUP – B

Part – I

Answer any one from the following : 5x1=5

1. *Write a program in C language to check an integer number whether it is even or odd. The number should be given as input except zero.*

Step 1. Step 1.Open the C programming IDLE (like Turbo C / DevC++) and type the given code in a new file:

```
/*2016 Part-1: Program1*/
#include<stdio.h>
void main()
   {int num;
   printf("\nEnter number (except zero): ");
   scanf("%d", &num);
   if(num%2==0)
       printf("\nNumber is even...");
   else
       printf("\nNumber is odd...");
   fflush(stdin);
   getchar();
   }
```

Step 2. Save the file as part1_P1.C and compile it.

Step 3. Run the program after compilation.

2. *Write a program in C language to take two strings as input and concatenate them.*

Step 1. Open the C programming IDLE and type the following code in a new file:

```
/*2016 Part-1: Program2*/
#include<stdio.h>
#include<string.h>
void main()
   {char str1[80], str2[80];
   printf("\nEnter first string: ");
   gets(str1);
   printf("\nEnter second string: "); fflush(stdin);
   gets(str2);
   strcat(str1, str2);
   printf("\nThe concatenated string is:\n");
   puts(str1);
   fflush(stdin);
   getchar();
   }
```

Step 2. Save the file as part1_P2.C and compile it.

Step 3. Run the program after compilation.

3. *Write a program in C language to check an integer number whether it is prime or not. The number will be taken as input.*

Step 1. Open the C programming IDLE and type the following code in a new file:

```
/*2016 Part-1: Program3*/
#include<stdio.h>
void main()
   {int num, i, flag=1;
   printf("\nEnter an integer: ");
   scanf("%d", &num);
   for (i=2; i<num; i++)
      {if(num%i == 0)
```

```
            {flag=0;
            break;
            }
        }
    if(flag==1)
        printf("\nPrime number...");
    else
        printf("\nNot a prime number...");
    fflush(stdin);
    getchar();
    }
```

Step 2. Save the file as part1_P3.C and compile it.

Step 3. Run the program after compilation.

4. ***Write a program in C language to make the following pattern:***

```
* * * *
* * *
* *
*
```

Step 1. Open the C programming IDLE and type the following code in a new file:

```
/*2016 Part-1: Program4*/
#include<stdio.h>
void main()
    {int i, j;
    for(i=4; i>=0; i--)
      {for(j=1; j<=i; j++)
        printf("* ");
      printf("\n");
      }
    fflush(stdin);
    getchar();
    }
```

Step 2. Save the file as part1_P4.C and compile it.

Step 3. Run the program after compilation.

Part – II

Answer any one from the following : 10x1=10

1. ***Write a program in C language which will take an array of 5 numbers and find out the largest and smallest numbers of that array.***

Step 1. Step 1. Open the C programming IDLE and type the following code in a new file:

```
/*2016 Part-2: Program1*/
#include<stdio.h>
void main()
    {int arr[5], i, max, min;
    for(i=0; i<5; i++)
        {printf("\nEnter array value[%d]: ", i)
        scanf("%d", &arr[i]);
        }
    max = arr[0];
    min = arr[0];
    for(i=0; i<5; i++)
        {if(arr[i]>max)
            max = arr[i];
        if(arr[i]<min)
```

```
            min = arr[i];
        }
    printf("\nMaximum = %d, Minimum = %d", max, min);
    fflush(stdin);
    getchar();
    }
```

Step 2. Save the file as part2_P1.C and compile it.

Step 3. Run the program after compilation.

2. ***Write a C program which can print the following series: 0, 1, 1, 2, 3, 5, ..., n (Fibonacci series). Value of n will be input from user.***

Step 1. Open the C programming IDLE and type the following code in a new file:

```
/*2016 Part-2: Program2*/
#include<stdio.h>
void main()
   {int i, num, t1=0, t2=1, t3=0;
   printf("\nEnter the last term value: ");
   scanf("%d", &num);
   while( t3 <= num )
      { printf("\n%d", t3);
      t1 = t2;
      t2 = t3;
      t3 = t1+t2;
      }
   fflush(stdin);
   getchar();
   }
```

Step 2. Save the file as part2_P2.C and compile it.

Step 3. Run the program after compilation.

3. ***Write a program in C language to convert all the lowercase letters in a sentence to the corresponding uppercase letters.***

Step 1. Open the C programming IDLE and type the following code in a new file:

```
/*2016 Part-2: Program3*/
#include<stdio.h>
void main()
   {char str[80];
   int i;
   printf("Enter sentence...");
   gets(str);
   for(i=0; str[i]!='\0'; i++)
      {if(str[i]>=97 && str[i]<=122)
         str[i] = str[i]-32;
      }
   printf("\nThe modified sentence is:\n%s", str);
   fflush(stdin);
   getchar();
   }
```

Step 2. Save the file as part2_P3.C and compile it.

Step 3. Run the program after compilation.

4. ***Write a program in C language to add two 3x3 matrices and store into another matrix and as well as display.***

Step 1. Open the C programming IDLE and type the following code in a new file:

```
/*2016 Part-2: Program4*/
#include<stdio.h>
```

```
int main()
   {int matA[3][3], matB[3][3], sum[3][3], i, j;
   for(i=0; i<3; i++)
      {for(j=0; j<3; j++)
         {printf("\nEnter value A[%d][%d]: ", i, j);
         scanf("%d", &matA[i][j]);
         }
      }

   for(i=0; i<3; i++)
      {for(j=0; j<3; j++)
         {printf("\nEnter value B[%d][%d]: ", i, j);
         scanf("%d", &matB[i][j]);
         }
      }

   printf("\nThe Sum Matrix...\n");
   for(i=0; i<3; i++)
      {for(j=0; j<3; j++)
         {sum[i][j] = matA[i][j] + matB[i][j];
         printf("\t%d", sum[i][j]);
         }
      printf("\n");
      }

   fflush(stdin);
   getchar();
   }
```

Step 2. Save the file as part2_P4.C and compile it.

Step 3. Run the program after compilation.

QUESTIONS OF ANNUAL EXAMINATION, 2016

COMPUTER SCIENCE

CLASS - XI

New Syllabus

Total Time : 3 Hours 15 minutes] [Full Marks : 70

Special credit will be given for answers which are brief and to the point.
Marks will be deducted for spelling mistakes, untidiness and bad handwriting.
Figures in the margin indicate full marks for the questions.

GROUP - A

1. Answer the following questions (MCQ type) : $1 \times 21 = 21$

(i) ROM is a
(a) Primary Memory (b) Secondary Memory
(c) Magnetic Memory (d) Flash Memory

(ii) Which of the following is not an Impact Printer?
(a) Inkjet Printer (b) Dot Matrix Printer
(c) Line Printer (d) Drum Printer

(iii) $(74)_{10} = (?)_2$
(a) $(111100)_2$ (b) $(1101010)_2$
(c) $(1001010)_2$ (d) None of these

(iv) 9's complement of $(123)_{10}$ is
(a) 867 (b) 786 (c) 876 (d) 678

(v) Which of the following gates is an Universal Gate?
(a) OR Gate (b) NOR Gate (c) XOR Gate (d) AND Gate

(vi) $A.\overline{A}.1 = ?$
(a) 0 (b) 1 (c) A (d) $\overline{A}$

(vii) How many selection lines are there in a 4×1 MUX ?
(a) 1 (b) 4 (c) 2 (d) 3

(viii) $A.(A + B) = ?$
(a) 0 (b) 1 (c) A (d) AB

(ix) The number of rows in the truth table of a Boolean formula involving four logical variables is
(a) 4 (b) 8 (c) 16 (d) 32

(x) The full form of POS is
(a) Product of Sum (b) Product or Sum (c) Product on Sum (d) None of these

(xi) The dual of the Boolean expression $A + \overline{C}$ is
(a) $A.\overline{C}$ (b) $\overline{A}.\overline{C}$ (c) AC (d) $\overline{A} + C$

(xii) To remove a file in Unix, which command is used ?
(a) delete (b) rm (c) del (d) rd

(xiii) One of the external commands of DOS is
(a) MOVE (b) DIR (c) DEL (d) COPY

(xiv) Which one of the following is not a keyword in *C* language ?
(a) do (b) goto (c) case (d) automatic

(xv) What type of operator is '&&' in *C* language ?
(a) Logical operator (b) Relational operator
(c) Arithmetic operator (d) Conditional operator

(xvi) Which of the following is used to represent the new line character in *C* language ?
(a) \ t (b) \ n (c) \ 0 (d) \ u

(xvii) Which header file is included for clrscr( ) and getch( ) functions ?
(a) math.h (b) stdio.h (c) stdlib.h (d) conio.h

(xviii) Which of the following is a Ternary operator ?
(a) ?& (b) :& (c) ?: (d) ?*

(xix) int a, b ;
$a = 5$, $b = 10$;
$a = ++ a + b ++$;
printf ("%d", a) ;
Write the correct output :
(a) 15 (b) 16 (c) 17 (d) 5

(xx) Which of the following is Exit controlled loop ?
(a) for loop (b) while loop (c) do-while loop (d) All of these

(xxi) Which of the following statements is used to create a structure ?
(a) struc (b) struct (c) structr (d) strct

GROUP – B

2. **Answer the following questions in brief (Alternatives are to be noted) :** $1 \times 14 = 14$

(i) What is the full form of MICR ?

OR

State one difference between Analog Computer and Digital Computer.

(ii) $(A12)_{16} = (?)_2$

OR

$(10110)_2 = (?)_6$

(iii) With the help of Boolean laws, prove that
$x + xy + xyz = x$.

OR

Simplify : $f = \overline{xy} + x\bar{y} + \bar{y}$

(iv) What do you mean by Multiprogramming ?

OR

Write a function of an Assembler.

(v) Write the names of two functions used for inputting character data in *C* language.

(vi) Write the statement to print 1 to 10 in *C* language.

OR

How will you create an array of 10 numbers ?

(vii) Write the syntax to make a user defined function in *C* language.

OR

Write a difference between Iterative function (Normal function) and Recursive function.

(viii) What is the function of pow( ) ?

OR

What is the function of sqrt( ) ?

(ix) What is the difference between '=' and '= =' in *C* language.

OR

Write down the differences between '*a*' and "*a*" in *C* language.

(x) Find out the output of the following *C* program :

```
int main( )
{
int x = 5, f = 1, i ;
for (i = 1 ; i < = x ; i ++ )
    {
  f = f * i ;
  }
printf ( "%d" , f) ;
return ( 0 ) ;
}
```

(xi) What are an operator and an operand in *C* language ?

OR

Write down the function of 'break' statement in *C* programming language.

(xii) In *C* language, what happens when condition is not given in the "for" statement ?

OR

What is Dummy or Empty loop ?

(xiii) Write two characteristics of Pointer.

(xiv) Write the significance of 'structure' in *C* language.

OR

Write one difference between '*a* & *b*' and '*a* && *b*'.

GROUP – C

3. Answer the following questions (Alternatives are to be noted) : $7 \times 5 = 35$

(i) (a) State any four differences between SRAM and DRAM.

(b) Write the functions of Data Bus and Address Bus.

(c) Write the significance of OMR. $2 + (2 + 2) + 1$

OR

(a) Write two features each of Super computer and Mainframe computer.

(b) Write two functions of Cache memory.

(c) What is Flash Memory ? $2 + (2 + 2) + 1$

(ii) (a) Add the following : $(234)_8 + (537)_8$.

(b) $(456.123)_8 = (?)_{16}$.

(c) Find out the 2's complement of $(-20)_{10}$. $2 + 2 + 3$

OR

(a) What is Signed Magnitude Number ? Give example.

(b) $(1110)_2 - (1010)_2$, using 2's complement do the subtraction.

(c) Multiply $(10110)_2$ with $(1101)_2$. $2 + 3 + 2$

(iii) (a) Write the truth table of a full adder and find out the logic expressions of it using *K*-map and also draw the logic diagram.

(b) Write two differences between Decoder and Encoder. $(2\frac{1}{2} + 2\frac{1}{2}) + 2$

OR

(a) Prove De Morgan's Theorem using truth table.

(b) Make the truth table of a half-subtractor and derive the expression for difference and borrow with using *K*-map.

(c) Make the truth table for the boolean function

$f(A, B, C) = \Sigma (0, 2, 4, 7)$

and also derive the logic expression for the same. $2 + 3 + 2$

(iv) (a) Define System Software with example.

(b) What are cold booting and warm booting ?

(c) Write the syntax of the following DOS Commands :

DIR, EDIT. $(2 + 1) + 2 + 2$

OR

(a) State two differences between high level language and low level language.

(b) Write two differences between Compiler and Interpreter.

(c) What is 'Shell' in UNIX ? Give example. $2 + 2 + (2 + 1)$

(v) (a) Write a *C* program to check an integer number whether it is prime or not taken from user as input.

(b) What is relational operator in *C* ? Discuss with the help of an example. $4 + 3$

OR

(a) Write a *C* program to print the following pattern :

```
*
* *
* * *
* * * *
```

(b) Write a *C* program to add the digits of a number. $4 + 3$

COMPUTER SCIENCE PAPER SOLUTION
CLASS - XI
2016 Annual Examination

GROUP – A (Multiple Choice Questions)

Question 1

(i) (a) Primary Memory
(ii) (a) Inkjet Printer
(iii) (c) 1001010_2
(iv) (c) 876
(v) (c) XOR Gate
(vi) (a) 0
(vii) (c) 2
(viii) (c) *A*
(ix) (c) 16
(x) (a) Product of Sum
(xi) (a) $A.\overline{C}$ [Correct answer should be $\overline{A}.C$. Option not given. closest answer is $A.\overline{C}$]
(xii) (b) rm
(xiii) (a) MOVE
(xiv) (d) automatic
(xv) (a) Logical operator
(xvi) (b) \n
(xvii) (d) conio.h
(xviii) (c) ? :
(xix) (b) 16
(xx) (c) do-while loop
(xxi) (b) struct

GROUP – B

Question 2

(i) Full form of MICR is Magnetic Ink Character Reader

OR

Analog computers store data as continuous voltage signals and digital computers store data as discrete voltage levels.

(ii) $(101000010010)_2$

(iii) $x + xy + xyz$
$= x\ (1 + y + yz)$
$= x.1$
$= x$

OR

$f = \overline{xy} + x\overline{y} + \overline{y}$
$= \overline{x} + \overline{y} + x\left(\overline{y} + 1\right)$
$= \overline{x} + \overline{y} + x$
$= \overline{x} + x + \overline{y}$
$= 1 + \overline{y}$
$= 1$

(iv) Multiprogramming means interleaved execution of two or more different independent jobs or programs by the same computer.

OR

An Assembler is used to translate a program in Assembly language to machine language.

(v) getch ( ), getchar ( )

(vi) for (i = 1; i < = 10 ; i + +)
printf ("\n%d", i) ;

OR

int arr [10] ;

(vii) return-data type function-name (paramater list)
{ function-body }

Alternative : int sum (int x_1, int y)
{ return x + y ;}

OR

A normal function runs once to process data and returns the result in general. A recursive function calls itself one or more times to process data and returns the result.

(viii) The pow ( ) function is used to calculate the result of x^y.
It is used as : Z = pow (x, y);

OR

The sqrt ( ) function is used to calculate the square root of a number. It is used as y = sqrt (x);

(ix) The '=' symbol represents an assignment operator and is used to assign the value of a variable or the result of a calculation to another variable.
The '= =' symbol represents the equality operator and is used to compare two values to see if they are same or not.

OR

'*a*' represents a character type data and is 1 byte in size. "*a*" represents a string type data and actually is 2 bytes in size as it stores the two characters '*a*' and '\ 0 '.

(x) 120

(xi) In the expression 5 + 3, 5 and 3 are the operands and '+' is the operator. Thus operators work on operands to give some result.

OR

The break statement is used to forcibly come out of a loop. It is used inside a loop along with an 'if' statement. It is also used to terminate a switch case.

(xii) When a condition is not given in a 'for' statement then the loop repeats infinitely *i.e.*, forms an infinite loop.

(xiii)
- A pointer is used to store the address of another variable
- An integer value can be added to a pointer

(xiv) A structure can be used in the C language to create an user defined data type. Example :
struct Point
{int x, y; };

OR

'*a & b*' indicates **bitwise** logical AND operation between the two values stored in the variables *a* and *b*.

'*a && b*' indicates logical AND operation between the two expressions *a* and *b*.

GROUP – C

Question 3

(i) (a)

SRAM	DRAM
1. No need to refresh at regular time interval	1. Needs to be refreshed at regular time intervals to retain data
2. Packing density is low	2. Packing density is high
3. Cost per bit is high	3. Cost per bit is low
4. Used to manufacture cache memory of a computer	4. Used to manufacture main memory of computer

(i) (b) **Data Bus** is concerned with transfer of data between CPU, memory and other hardware locations. A 32 bit data bus can transfer 4 bytes of data at a time.
Address Bus connects the CPU and the RAM and carries the memory address to activate a certain memory location in the RAM.

(c) OMR stands for Optical Mark Reader. It is an input device used for optical mark recognition *i.e.*, to read data from pre-printed document forms marked by humans. It is used in areas like reading answer scripts with multiple choice questions.
A user inputs the information by darkening circles marked on a pre-printed sheet using pencil. The OMR device then automatically reads the marked circles.

OR

(i) (a) Two features of super computers :
- Used mainly in Scientific and Research organisations for processing complex scientific data that require a huge processing power
- The cost of a supercomputer is much higher compared to an ordinary computer

Two features of mainframe computers :
- Mainframe computers can support several thousand users simultaneously *i.e.*, are multiuser systems.
- Mainly used in organisations that need to frequently access the same information and quickly process large number of transactions on-line.

(b) Two functions of cache memory are :
- It is a high speed memory that is used to temporarily store active data and instructions during data processing.
- Cache memory increases the performance of the PC

(c) Flash memory is a solid state secondary memory storage device. It works similar to an EEPROM and uses a special CMOS transistor for storing data. Data can be stored, read, and erased from such a memory. It is used in various applications like computer BIOS chip, pendrives, smart media etc.

(iii) (a) $(234)_8 + (537)_8$

$$\begin{array}{r} {\scriptstyle 1} \\ 2\ 3\ 4_8 \\ +\ 5\ 3\ 7_8 \\ \hline 7\ 7\ 3_8 \end{array}$$

Alternative Process $\Rightarrow$

$$\begin{array}{r} {\scriptstyle 1} \\ 2\ 3\ 4_8 \\ +\ 5\ 3\ 7_8 \\ \hline 7\ 7\ 11 \\ -\ 8 \\ \hline 3 \end{array}$$

Ans. $(773)_8$

Ans. $(773)_8$

(b)

$$(4\ 5\ 6\ .\ 1\ 2\ 3)_8$$

$$\underbrace{0\,0\,0\,1}_{1}\ \underbrace{0\,0\,1\,0}_{2}\ \underbrace{1\,1\,1\,0}_{E}\ .\ \underbrace{0\,0\,1\,0}_{2}\ \underbrace{1\,0\,0\,1}_{9}\ \underbrace{1\,0\,0\,0}_{8}$$

Ans. $(1\ 2\ E\ .\ 2\ 9\ 8)_{16}$

(c)

		rem
2 | 20		0
2 | 10		0
2 | 5		1
2 | 2		0
2 | 1		1
0		

$-20_{10} = -10100_2$

In **8 bit** form 0 0 0 1 0 1 0 0 [+20 in sign magnitude]

↓ 1's complement

1 1 1 0 1 0 1 1 [−20 in 1's complement form]

↓ + 1

1 1 1 0 1 1 0 0 [−20 in 2's complement form]

(ii) OR

(a) In a Sign Magnitude Number the MSB of the number represents the sign of the number and the remaining bits represent the magnitude of the number. If the number is +ve, the sign bit is taken as 0. If the number is –ve, the sign bit is taken as 1.

Example : $+10111_2$ in sign magnitude representation in 8 bits

= 0 0 0 1 0 1 1 1

– 1 0 1 1 1 in sign magnitude representation in 8 bits

= 1 0 0 1 0 1 1 1

Sign bit Magnitude

(b) $1110_2 - 1010$

= 1110 + (2's complement of − 1010)

= 1110 + 0110

```
  1  1
  1  1  1  0
+ 0  1  1  0
-------------
✗ 0  1  0  0        Ans. (100)2
```

R.W.

1 0 1 0

↓ 1's complement

0 1 0 1

+ 1

0 1 1 0 (2's complement)

(c)

```
          1  0  1  1  0
        ×  1  1  0  1
-----------------------
          1  0  1  1  0
    1  0  0  0  0  0  ×
 1  1  0  1  1  0  ×  ×
 1  1  0  1  1  0  ×  ×  ×
-----------------------
 1  0  0  0  1  1  1  1  0     Ans. (100011110)2
```

(iii) (a) Truth Table of Full Adder

A	B	C	Sum	Cout	Min sum	Min Cout
0	0	0	0	0		
0	0	1	1	0	$\overline{A}\,\overline{B}\,C$	
0	1	0	1	0	$\overline{A}\,B\,\overline{C}$	
0	1	1	0	1		$\overline{A}\,B\,C$
1	0	0	1	0	$A\,\overline{B}\,\overline{C}$	
1	0	1	0	1		$A\,\overline{B}\,C$
1	1	0	0	1		$A\,B\,\overline{C}$
1	1	1	1	1	$A\,B\,C$	$A\,B\,C$

Sum = $\overline{A}\,\overline{B}\,C + \overline{A}\,B\,\overline{C} + A\,\overline{B}\,\overline{C} + ABC$

Cout = $\overline{A}\,B\,C + A\,\overline{B}\,C + A\,B\,\overline{C} + ABC$

Sum

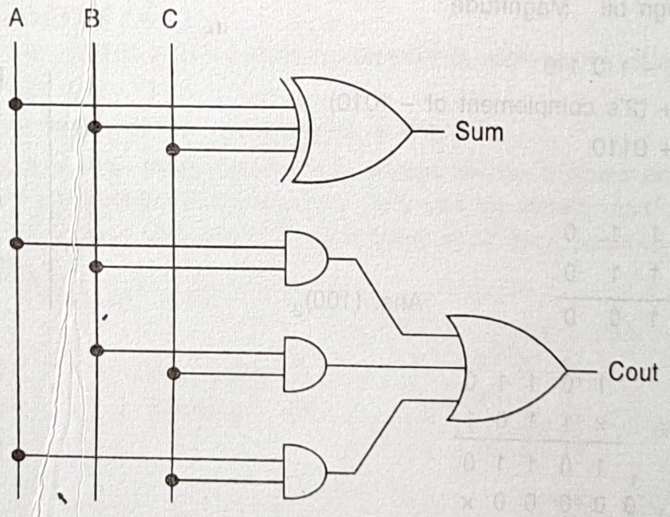
A \ BC	$\overline{B}\,\overline{C}$	$\overline{B}\,C$	BC	$B\,\overline{C}$
$\overline{A}$		1		1
A	1		1	

Cout

A \ BC	$\overline{B}\,\overline{C}$	$\overline{B}\,C$	BC	$B\,\overline{C}$
$\overline{A}$			1	
A		1	1	1

$S = \overline{A}\,\overline{B}\,C + \overline{A}\,B\,\overline{C} + A\,\overline{B}\,\overline{C} + ABC$ $\quad Cout = AB + BC + AC$

$= A \oplus B \oplus C$



Circuit for Full Adder.

(b) **DIFFERENCES**

Decoder	Encoder
1. Converts binary data to decimal or other base	1. Converts from other base or decimal to binary.
2. An *n*-to-*m* decoder decodes *n* input bits to $m = 2^n$ output bits	2. An *n*-to-*m* encoder can encode at the most $n = 2^m$ numbers to *m* binary bits.

(iii) OR

(a)

1	2	3	4	5	6	7	8	9	10
A	B	$\overline{A}$	$\overline{B}$	$A+B$	$\overline{A+B}$	$\overline{A}.\overline{B}$	$A.B$	$\overline{A.B}$	$\overline{A}+\overline{B}$
0	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	0	0	1	1
1	0	0	1	1	0	0	0	1	1
1	1	0	0	1	0	0	1	0	0

From columns 6 and 7 we find that $\overline{A+B} = \overline{A}.\overline{B}$

From columns 9 and 10 we find that $\overline{A.B} = \overline{A}+\overline{B}$

(b) **Half Subtractor**

x	y	diff	borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Diff. K′ Map

x \ y	$\overline{y}$	y
$\overline{x}$	0	1
x	2	3

Borrow K′ Map

x \ y	$\overline{y}$	y
$\overline{x}$	0	1
x	2	3

$D = \overline{x}y + x\overline{y}$ $B = \overline{x}y$

(c) $f = \Sigma(0, 2, 4, 7)$

A	B	C	f	Min
0	0	0	1	$\overline{A}\,\overline{B}\,\overline{C}$
0	0	1	0	
0	1	0	1	$\overline{A}\,B\,\overline{C}$
0	1	1	0	
1	0	0	1	$A\,\overline{B}\,\overline{C}$
1	0	1	0	
1	1	0	0	
1	1	1	1	$A\,B\,C$

$f = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,B\,\overline{C} + A\,\overline{B}\,\overline{C} + A\,B\,C$

(iv) (a) System software include software that are used to run a computer and manage its different resources. These software interact with the hardware to make it work. The operating system is the most important system software. It acts as a link between the hardware and the user.

(b) When the computer is powered on then the booting that takes place is called cold booting. It involves the power on Self Test to check the RAM, I/O Devices and the microprocessor followed by loading the operating system.

During warm booting the computer is restarted using the Ctrl + Alt + Del key combination. Warm booting skips the Power On Self Test (POST) and loads the operating system files.

(c) DIR ⇒ C : \ > DIR C : \ Temp

EDIT ⇒ C : \ > EDIT C : \ Temp \ Myfile.txt

OR

(a) High Level Language	Low Level Language
1. Language more close to human language	1. Language more close to machine or hardware
2. It is easier to write programs using a high level language	2. It is difficult to write programs using a low level language.
3. Example : C, C++, BASIC	3. Example : Machine language.

(b) Compiler	Interpreter
1. It translates the entire program before it is run	1. It translates the program line-by-line and runs each line
2. A compiled program has to be compiled only once and run as many times required.	2. An interpreted program needs to be interpreted everytime it is run.

(c) The shell serves as the user interface through which the user interacts with the operating system. It is the layer next to the Kernal. It acts as a command interpreter and manages the running of application programs.

Examples of different UNIX shells include the Bourne Shell, the Korn Shell.

(v) (a)
```
#include < stdio.h >
void main ( )
{ int num, i, flag = 1;
    printf ("\nEnter number :");
    scanf ("%d", &num);
    for (i = 2; i < num; i++)
        { if (num% i = = 0)
            { flag = 0;
            break;
            }
        }
if ( flag = = 1)
    printf ("\n Prime number...");
else
    printf ("\n Not a prime number...");
}
```

(b) Relational operators are used to compare two values in C. There are six different relational operators. The outcome of a relational operator is either True or False *i.e.*, produce a logical output. The different relational operators are. < , < = , > , > = , = = , ! =

Example :

```
int num ;
printf ("\n Enter a number :") ;
scanf ("%d", &num) ;
if (num % 2 = = 0)
    printf ("\n Even number") ;
else
    printf ("\n Odd number") ;
```

The above code compares the result of num %2 with the value '0' for equality. If the remainder is '0' then the result is True. If the remainder is '1' then the result of comparison is False.

OR (a)
```
# include < stdio.h >
void main ( )
  { int i, j, k,
    for ( i = 1 ; i < = 4 ; i ++ )
      { for (j = 1; j < = 4 – i ; j ++ )
          printf ( "ø" ) ;
        for (k = 1 ; k < = i ; K ++ )
          printf ( " * ø " ) ;
        printf ("\n") ;
      }
  }
```

Output



[Note : Here ø indicates a blank space]

(b)
```
# include < stdio.h >
void main ( )
  { int num, digit, sum = 0 ;
    printf ("\n Enter number ...") ;
    scanf ("%d", &num);
    while ( num > 0 )
      { digit = num % 10;
        sum = sum + digit ;
        num = num / 10 ;
      }
    printf ("\n The required sum = %d", sum ) ;
  }
```

QUESTIONS OF ANNUAL EXAMINATION, 2017
COMPUTER SCIENCE
CLASS – XI
(PRACTICAL)
(NEW SYLLABUS)

[Time : 3 hours] [Full Marks : 30]

Special credit will be given for answers which are brief and to the point.
Marks will be deducted for spelling mistakes, untidiness and bad handwriting.
Figures in the margin indicate full marks for the questions.

General Instructions :

- There are 4 (four) groups. Group-A contains 4 (four) questions. Group-B contains 2 (two) parts, each containing 4 (four) questions. You have to answer 1 (one) question from Group-A and 1 (one) question from each part of Group-B.
- Write all the steps in your answer-script which you have performed with the computer.
- Print all files, if necessary and possible, otherwise write all files with partial data input and output in your answer-script.
- Make suitable assumptions, if any, and tabulate them.

GROUP – A

Answer any *one* from the following questions : 5 x 1 = 5

1. Carry out the following using the Windows operating system :
   (a) Make three folders in 'C' drive and give their names as 'Science', 'Arts', and 'Commerce'.
   (b) Create a shortcut of 'Science' folder in the Desktop.
   (c) Make a folder named as 'New Folder' inside the 'Arts' folder and then move it into the 'Commerce' folder. 2 + 1 + 2
2. (a) Create a folder of your name in the 'D' drive of your computer. After that, rename the folder as 'STUDENT'.
   (b) Set Screensaver of your choice in your computer and set the waiting time as 30 minutes.
   (c) Change the taskbar location of MS-Windows of your computer to the right side of the screen. 2 + 2 + 1
3. Do the following tasks using Unix / Linux command :
   (a) Using Editor, make a file named as 'Sample file'.
   (b) OFF the 'Write' permission of that file.
   (c) Move that file to inside of any folder. 2 + 2 + 1
4. (a) Show the list of all current users using Unix/Linux command.
   (b) Using Unix/Linux command, create a directory 'XI-Final' under your current directory.
   (c) Using Unix/Linux command, rename the folder 'XI-Final' as 'Xi-Lab'. 2 + 2 + 1

GROUP – B

PART – I

Answer any *one* from the following questions : 5 x 1 = 5

- Algorithm / Flowchart : 2
- Program coding in 'C' language : 2
- Execution and output : 1

1. Write a program in 'C' language to find out the summation of 1 to *n* integers (value of *n* should be taken as input). 5

2. Write a program in 'C' language in which digits of an inputted number are printed in reverse.

3. By using 'C' language write a program which will convert an inputted text / word into uppercase. The program should also display the converted text. 5

4. Write a program in 'C' language to make the following pattern :

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

5

PART – II

Answer any *one* from the following questions : 10 x 1 = 10

- Algorithm / Flowchart : 4
- Program coding in 'C' language : 4
- Execution and output : 2

1. By using 'C' programming language write a program which will display all the prime numbers between 1 to 100. 10

2. Write a program in 'C' to convert a decimal integer to its equivalent Binary. The decimal integer should be fed through keyboard. 10

3. Write a program in 'C' language to find out the x^n using recursive function where *x* and *n* are two integers, should be taken as input. 10

4. Write a 'C' program which can accept a sentence from user, count the vowels in it and show the total number of vowels present in that sentence. 10

GROUP – C

Laboratory notebook 5

GROUP – D

Viva-voce 5

COMPUTER SCIENCE PAPER SOLUTION
CLASS - XI
2017 Practical Examination

GROUP – A (Multiple Choice Questions)

Answer any one from the following : 5×1=5

1. *Carry out the following using the Windows operating System:*

   (a) Make three folders in 'C' drive and give their names as 'Science', 'Arts', and 'Commerce'.

   Step 1. Double click on the Computer icon on the Desktop
   Step 2. In the window that opens showing the different drive names, double click on the C: drive icon
   Step 3. In the window that opens, right click on the white space in the right side pane
   Step 4. In the dropdown list that opens, select the New option
   Step 5. In the sub-list that opens, click on the Folder option to create a New Folder
   Step 6. Type the name of the folder as 'Science' and click anywhere outside the folder name
   Step 7. Repeat steps 3 to 6 to create two more new folders and name them as 'Arts' and 'Commerce'.

   (b) Create a shortcut of the 'Science' folder in the Desktop.

   Step 1. Right click on the Science folder and select the Send to option from the dropdown list.
   Step 2. In the sub-list that opens, select the Desktop (create shortcut) option to create a shortcut of the 'Science' folder on the Desktop.

   (c) Make a folder named as 'New Folder' inside the 'Arts' folder and then move it into the 'Commerce' folder.

   Step 1. Double click on the newly created 'Arts' folder to open the 'Arts' folder window.
   Step 2. In the window that opens, right click on the white space in the right side pane
   Step 3. In the dropdown list that opens, select the New option
   Step 4. In the sub-list that opens, click on the Folder option to create a New Folder
   Step 5. Click anywhere outside the folder name to create the folder
   Step 6. Right click on the 'New Folder' icon and select the Cut option from the dropdown list
   Step 7. Click on the 'Back to Local Disk (C:)' blue arrow button at the top left corner of the window to get back to the C drive.
   Step 8. Double click on the 'Commerce' folder name to open the 'Commerce' folder.
   Step 9. Right click on the right side window pane and select the Paste option to move the New Folder from Arts to Commerce folder

2. *(a) Create a folder of your name in the D: drive of your computer. After that, rename the folder as 'STUDENT'.* (2)

   Step 1. Double click on the Computer icon on the Desktop
   Step 2. In the window that opens showing the different drive names, double click on the D: drive icon
   Step 3. In the window that opens, right click on the white space in the right side pane
   Step 4. In the dropdown list that opens, select the **New** option
   Step 5. In the sub-list that opens, click on the Folder option to create a **New Folder**
   Step 6. Type the name of the folder as 'Avik' (i.e. type your own name here) and click anywhere outside the folder name
   Step 7. Right click on the newly created folder name. From the dropdown list that opens select the Rename option
   Step 8. Re-type the folder name as 'STUDENT' and click outside the folder icon

   (b) Set Screensaver of your choice in your computer and set the waiting time as 30 minutes. (2)

   Step 1. Right click on the **Desktop** and click on the **Personalize** option from the dropdown list that appears
   Step 2. From the **Personalization** window that opens, select the **Screen Saver** option from the bottom

Step 3. The Screen saver window opens. Select the **Bubbles** option from the Screen saver list box

Step 4. Under the **Wait Time** box type the time as 30

Step 5. Click on **Apply** to apply the screen saver

(c) *Change the taskbar location of MS Windows of your computer to the right side of the screen.* (1)

Step 1. Right click on the **Taskbar** at the bottom of the screen

Step 2. Choose the **Properties** option from the drop down list that appears

Step 3. The Taskbar and Start Menu Properties window opens

Step 4. Under the **Taskbar appearance section**, click on the list beside the Taskbar location on screen

Step 5. Select the Right option and click on the Apply button to move the Taskbar to the right of the screen

3. ***Do the following tasks using UNIX / Linux command:*** (2)

(a) *Using Editor, make a file named as 'Sample file'.*

Step 1. Open the vi editor in UNIX to create the file 'Sample file' by typing the following:
$ vi SampleFile

Step 2. Type i to type any text in the file

Step 3. Press Enter followed by Esc to get back to the command mode

Step 4. To save the file, type : **wq!** And press Enter to save and exit the vi editor

(b) *OFF the 'Write' permission of that file.* (2)

Step 1. Type the following under the command prompt:
$ chmod 555 SampleFile

(c) *Move the file to inside of any folder.* (1)

Step 1. Type the following under the command prompt to move the file to the usr directory
$ mv Sample File / usr

4. *(a) Show the list of all current users using Unix/Linux command.* (2)

Step 1. Type the following command to display the current users:
$ who

(b) *Using Unix/Linux command, create a directory 'XI-Final' under your current directory.* (2)

Step 1. Type the following under the command prompt:
$ mkdir XI-Final

(c) *Using Unix/Linux command rename the folder 'XI-Final' as 'XI-Lab'.* (1)

Step 1. Type the following under the command prompt to move the file to the usr directory
$ mv XI-Final XI-Lab

GROUP – B

Part – 1

Answer any one from the following : 5x1=5

1. *Write a program in C language to find out the summation of 1 to n integers (value of n should be taken as input):*

Step 1. Open the C programming IDLE (like Turbo C or DevC++) and type the following code in a new file:

```
/*2017 Part-1: Program1*/
#include<stdio.h>
int main()
   {int n, i, sum=0;
   printf("\nEnter any integer: ");
   scanf("%d", &n);
   for(i=1; i<=n; i++)
      sum = sum + i;
   printf("\nThe required sum is %d", sum);
   fflush(stdin);
```

```
getchar();
return 0;
}
```

Step 2. Save the file as part1_P1.C and compile it.

Step 3. Run the program after compilation.

2. *Write a program in C language in which digits of an input number are printed in reverse.*

Step 1. Open the C programming IDLE and type the following code in a new file:

```
/*2017 Part-1: Program2*/
#include<stdio.h>
int main()
   {int num, digit;
   printf("\nEnter any integer: ");
   scanf("%d", &num);
   while(num>0)
      {digit = num%10;
      printf("%d", digit);
      num = num/10;
      }
   fflush(stdin);
   getchar();
   return 0;
   }
```

Step 2. Save the file as part1_P2.C and compile it.

Step 3. Run the program after compilation.

3. *By using C language write a program which will convert an input text/word into uppercase. The program should also display the converted text.*

Step 1. Open the C programming IDLE and type the following code in a new file:

```
/*2017 Part-1: Program3*/
#include<stdio.h>
int main()
   {int i;
   char sent[80];
   printf("\nEnter a word or sentence: ");
   gets(sent);
   for(i=0; sent[i]!='\0'; i++)
      {if( sent[i] >=97 && sent[i] <= 122)
         sent[i] = sent[i] - 32;
      }
   printf("\nThe modified text is:\n%s", sent);
   fflush(stdin);
   getchar();
   return 0;
   }
```

Step 2. Save the file as part1_P3.C and compile it.

Step 3. Run the program after compilation.

4. *Write a program in C language to make the following pattern:*

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Step 1. Open the C programming IDLE and type the following code in a new file:

```
/*2017 Part-1: Program4*/
#include<stdio.h>
int main()
   {int i, j;
```

```
        for(i=1; i<=k; i++)
        {for(j=1; j<=i; j++)
           printf("%d ", j);
        printf("\n");
        }
     fflush(stdin);
     getchar();
     return 0;
     }
```

Step 2. Save the file as part1_P4.C and compile it.

Step 3. Run the program after compilation.

GROUP – B

Part – 2

Answer any one from the following : 10x1=10

1. By using 'C' programming language write a program which will display all the prime numbers between 1 to 100:

   Step 1. Open the C programming IDLE and type the following code in a new file:

```
/*2017 Part-2: Program1*/
#include<stdio.h>
int main()
   {int num, i, flag;
   for(num=1; num<=100; num++)
      {flag=1;
      for(i=2; i<=num/2; i++)
         {if(num%i == 0)
            {flag=0;
            break;
            }
         }
      if(flag==1)
         printf("\n%d", num);
      }
   fflush(stdin);
   getchar();
   return 0;
   }
```

   Step 2. Save the file as part2_P1.C and compile it.

   Step 3. Run the program after compilation.

2. *Write a program in 'C' to convert a decimal integer to its equivalent Binary. The decimal integer should be fed through keyboard:*

   Step 1. Open the C programming IDLE and type the following code in a new file:

```
/*2017 Part-2: Program2*/
#include<stdio.h>
int main()
   {int num, i=0, digit, bin[32], j, binNum=0;
   printf("\nEnter any integer: ");
   scanf("%d", &num);
   while(num>0)
      {digit = num%2;
      bin[i] = digit;
      num = num/2;
      i++;
      }
   for(j=i-1; j>=0; j--)
      binNum = binNum*10 + bin[j];
   printf("\nBinary number = %d", binNum);
```

```
fflush(stdin);
getchar();
return 0;
}
```

Step 2. Save the file as part2_P2.C and compile it.
Step 3. Run the program after compilation.

3. *Write a program in 'C' language to find out x^n using recursive function where x and n are two integers, should be taken as input.*

Step 1. Open the C programming IDLE and type the following code in a new file:

```
/*2017 Part-2: Program3*/
#include<stdio.h>
int power(int x, int n)
   {int p;
   if(n==0)
      return 1;
   p = x * power(x, n-1);
   return p;
   }
int main()
   {int n, x, pow;
   printf("\nEnter base value: ");
   scanf("%d", &x);
   printf("\nEnter index value: ");
   scanf("%d", &n);
   pow = power(x,n);
   printf("\nRequired power = %d", pow);
   fflush(stdin);
   getchar();
   return 0;
   }
```

Step 2. Save the file as part2_P3.C and compile it.
Step 3. Run the program after compilation.

4. *Write a 'C' program which can accept a sentence from the user, count the vowels in it and show the total number of vowels present in that sentence.*

Step 1. Open the C programming IDLE and type the following code in a new file:

```
/*2017 Part-2: Program4*/
#include<stdio.h>
int main()
   {int i, j, count=0;
   char sent[80], vowel[] = "AEIOUaeiou";
   printf("\nEnter a sentence: ");
   gets(sent);
   for(i=0; sent[i]!='\0'; i++)
      {for(j=0; vowel[j]!='\0'; j++)
         if(sent[i] == vowel[j])
            {count++;
            break;
            }
      }
   printf("\nTotal vowels = %d", count);
   fflush(stdin);
   getchar();
   return 0;
   }
```

Step 2. Save the file as part2_P4.C and compile it.
Step 3. Run the program after compilation.

QUESTIONS OF ANNUAL EXAMINATION, 2017

COMPUTER SCIENCE

CLASS – XI

New Syllabus

Total Time : 3 Hours 15 minutes] [Full Marks : 70

Special credit will be given for answers which are brief and to the point.
Marks will be deducted for spelling mistakes, untidiness and bad handwriting.
Figures in the margin indicate full marks for the questions.

GROUP – A

1. **Answer the following questions (MCQ type) :** $1 \times 21 = 21$

(i) The technology of third generation computer is

(a) Vacuum Tube (b) LSI (c) IC (d) VLSI

(ii) Which of the following is not an Input Device ?

(a) OCR (b) CRT (c) OMR (d) MICR

(iii) $(A)_{16} + (5)_{16} = (?)_{16}$

(a) 15 (b) F (c) E (d) D

(iv) The BCD of $(24)_{10}$ is

(a) 10100 (b) 00100100 (c) 101001 (d) 10010100

(v) 'The complement of product of two variables is equal to the summation of their individual complements.'

Which of the following operation satisfies the statement ?

(a) $\overline{A.B} = \bar{A}.\bar{B}$ (b) $\overline{A.B} = \bar{A} + \bar{B}$ (c) $\bar{A}.\bar{B} = \overline{A+B}$ (d) $\overline{A+B} = \bar{A} + \bar{B}$

(vi) How many NAND gates are required to form an OR gate using NAND gate ?

(a) 3 (b) 4 (c) 2 (d) 5

(vii) $\overline{\bar{x}+\bar{y}}$ = ?

(a) $x+y$ (b) $\bar{x}+\bar{y}$ (c) $x.y$ (d) $\bar{x}.\bar{y}$

(viii) How many values can be placed at most in a three variable K-Map ?

(a) 7 (b) 8 (c) 3 (d) 6

(ix) Maxterm is denoted by

(a) Σ (b) Ω (c) θ (d) π

(x) In Boolean algebra, $F(a, b, c) = \Sigma(0, 4, 7)$ is same as

(a) $\pi(1, 2, 3, 5, 6)$ (b) $\pi(0, 4, 7)$ (c) $\pi(0, 2, 4, 6)$ (d) $\pi(1, 3, 5, 7)$

(xi) In Boolean Algebra, $AC + A\bar{B}C$ is equal to

(a) AB (b) BC (c) AC (d) $A + BC$

(xii) Which of the following is not a GUI operating system ?

(a) MS-DOS (b) MS-Windows (c) UNIX (d) None of these

(xiii) The command to make a file in UNIX operating system is

(a) mkdir (b) cat (c) copy con (d) ls

(xiv) Which of the following is not a primary data type in C ?

(a) float (b) enum (c) char (d) double

(xv) Which of the following is a valid variable name in C language ?

(a) $a-1$ (b) $1a$ (c) al (d) $1-a$

(xvi) Which of the following is equivalent to if-else statement ?

(a) ?* (b) ?: (c) ?$ (d) ?#

(xvii) Which of the following is a valid array in *C* language ?

(a) [*a*] (b) *a* ( ) (c) *a* [5] (d) *a* [-]

(xviii) In *C* language which of the following is a user defined function ?

(a) printf( ) (b) scanf( ) (c) gets( ) (d) main( )

(xix) main( )

{ int *a* = 1, *b* = 5;

while (*a* ++ < 5)

{ b++ ;

}

printf { "%*d*", *b*) ;

getch( ) ;

}

Here the value of *b* will be

(a) 7 (b) 8 (c) 9 (d) 10

(xx) main( )

{ int *i* ;

for (*i* = 1 ; *i* < = 5 ; *i* ++)

printf ("%*d*", *i*) ;

i++ ;

getch( ) ;

}

Here the output will be

(a) 1234 (b) 12345 (c) 2345 (d) 234

(xxi) $(23)_{10}$ ^ $(29)_{10}$ = $(?)_{10}$

(a) $(10)_{10}$ (b) $(19)_{10}$ (c) $(6)_{10}$ (d) 0

GROUP – B

2. **Answer the following questions in brief (Alternatives are to be noted) :** 1 × 14 = 14

(i) What do you mean by data processing ? Give example.

Or, Write two features of Non-impact Printer.

(ii) $(74)_8 - (47)_8 = (?)_8$.

Or, $(1011.10)_2 \times (11)_2 = (?)_2$

(iii) What is Don't Care Condition in Boolean Algebra ?

Or, Prove that, $x + 1 = 1$.

(iv) What is spooling ?

Or, What do you mean by buffering ?

(v) What is Algorithm ?

Or, Write two features of flowchart.

(vi) Write two difference between while and do while loop.

(vii) What is the role of getch( ) in *C* language ?

(viii) Write the significance of Header file.

Or, What is Library function ?

(ix) Write the Syntax of else-if ladder in *C* language.

(x) Write the Syntax to take the data elements as input in a 2 × 3 array.

Or, How can you declare a string variable ? Show by an example.

(xi) int $a = 0, i$;
for ($i = 1$; $i <= 5$; i++)
{ $a = a + i$;
}
printf ("%d", a) ;
Find out the value of a.

(xii) Give an example of an Infinite loop.

(xiii) What are the different storage classes in C language ?
Or, What is Recursion ?

(xiv) What do you mean by 'a++' and '++a' ?

GROUP – C

3. **Answer the following questions (Alternatives are to be noted) :** $7 \times 5 = 35$

(i) (a) Write two functions of ALU.
(b) Discuss Laser Printer in short.
(c) Write a short note on Barcode Reader. 2 + 3 + 2

Or, (a) Write three characteristics of second generation computers.
(b) Write two differences between Data and information.
(c) Write a short note on DVD. 3 + 2 + 2

(ii) (a) Subtract the following : $(AB6E)_{16} - (C2DF)_{16}$.
(b) $(12A7.13)_{16} = (?)_2$
(c) $(1011)_2 + (1101)_2 = (?)_2$. 3 + 2 + 2

Or, (a) Subtract the following using 1's complement method :
$(1101)_2 - (1001)_2$.
(b) What is ASCII code ? Give example.
(c) Find out the Gray code of $(1101)_2$. 3 + 2 + 2

(iii) (a) Why NOR gate is called universal gate ? Prove it.
(b) Write down the truth table of full substractor and draw the logic diagram of it.
(c) Simplify :
$F = (A + \overline{B} + C).(\overline{A} + \overline{B} + C).(A + \overline{B} + \overline{C}).(\overline{A} + \overline{B} + \overline{C})$ 2 + 3 + 2

Or, (a) Draw the block diagram of 4 × 1 MUX.
(b) $F = (A + \overline{B}).(\overline{A} + B)$, find out the $\overline{F}$.
(c) Prove that SOP = $\overline{POS}$ (Show by an example). 2 + 2 + 3

(iv) (a) What is time sharing OS ? Give example.
(b) What are the functions of virtual memory ?
(c) Write down the function of the MS-DOS 'ATTRIB' command with example.

Or, (a) What do you mean by batch file in MS-DOS ?
(b) What is Multiprocessing OS ? Give example.
(c) Give the example of Wild Card character in UNIX OS and also write the advantages. 2 + 3 + 2

(v) (a) What do you mean by Entry controlled loop in C language ? Give example.
(b) $(25)_{10}$ & $(19)_{10} = (?)_{10}$
(c) Write a program in C language to check a number is Palindrome or not. 2 + 1 + 4

Or, (a) Write the Syntax of Nested if statement.
(b) $(21)_{10} \mid (19)_{10} = (?)_{10}$.
(c) Write a Program in C language to print the following pattern : 2 + 1 + 4

COMPUTER SCIENCE PAPER SOLUTION
CLASS – XI
2017 Annual Examination

GROUP – A

1. **Answer the following questions (MCQ type) :**

(i) (c) IC
(ii) (b) CRT
(iii) (b) F
(iv) (b) 00100100
(v) (b) $\overline{A.B} = \overline{A} + \overline{B}$
(vi) (a) 3
(vii) (c) *x.y*
(viii) (b) 8
(ix) (d) π
(x) (a) π(1, 2, 3, 5, 6)
(xi) (c) AC
(xii) (a) MS-DOS
(xiii) (b) cat
(xiv) (b) enum
(xv) (c) *a*1
(xvi) (b) ?:
(xvii) (c) a[5]
(xviii) (d) main( )
(xix) (c) 9
(xx) (b) 12345
(xxi) (a) $(10)_{10}$ [Not in syllabus]

GROUP – B

2. **Answer the following questions in brief (Alternatives are to be noted) :**

(i) Data processing by a computer involves input, process, and output. The input part is used to input the data, process part involves modification of the data and the output part involves outputting the processed data.

OR

Two features of non-impact printers are :

1. Non-impact printers do not have any hammering mechanism to apply ink on paper
2. Non-impact printers are much less noisy than impact printers.

(ii)
$$\begin{array}{r} {}^{6}\not{7}\ 14_8 \\ -4\ \ 7_8 \\ \hline 2\ \ 5_8 \end{array}$$

OR
$$\begin{array}{r} 1011.10 \\ \times\ 11 \\ \hline '1'0'1110 \\ '101110\times \\ \hline 100010.10_2 \end{array}$$

Ans : $(100010.10)_2$

(iii) In case a particular input combination does not occur in a truth table, then the output of such an input combination is taken as don't care term.

OR

x	$x + 1$
0	0 + 1 = 1
1	1 + 1 = 1

(iv) The full form of spooling is simultaneous peripheral operations online. It is a technique used to solve the problem of speed mismatch between the processor and the peripheral devices like printers, keyboards etc.

OR

Buffering involves storing data temporarily while it is moved from one place to another. Buffers are used when there is a difference between the rate at which the data is received and the rate at which it can be processed.

(v) An algorithm is a sequence of instructions to obtain a solution to a problem in a finite number of steps.

OR

Two features of flowcharts are :

1. A flowchart gives a graphical representation of an algorithm with special symbols representing different operations.
2. The same flowchart can be used with various programming languages *i.e.* a flowchart is independent of any programming language.

(vi)

Difference	
while	do-while
1. The loop condition is checked at the beginning	1. The loop condition is checked at the end.
2. The loop body may or may not run at all depending upon the condition.	2. The loop body runs at least once.

(vii) The getch( ) function is used to get a character data from the keyboard. It can be used to pause the program also.

(viii) Header files are required when library functions are used in a C program. The header files provide the prototype of the library functions used.

OR

A library function is predefined and precompiled piece of code that can be used in a C program to carry out various specific operations.

(ix) Syntax of else – if ladder :

```
if ( x = = 1)
      printf ("Value = 1") ;
else if (x = = 2)
      printf ("\n Value = 2") ;
else if (x = = 3)
      printf ("\n Value = 3") ;
else
      printf ("\n Any other value") ;
```

```
if (condition - 1)
      statement;
else if (condition - 2)
      statement;
else if (condition - 3)
      statement;
else
      statement;
```

(x)
```
for  (i = 0 ; i < 2 ; i++)
     { for ( j = 0; j < 3 ; j++)
          { printf("\nEnter value : ");
          scanf("%d", & mat [ i ] [ j ];
          }
     }
```

OR

A string variable is declared in the following manner : char str [80];

(xi) Ans : 15

(xii) Example of an infinite loop :

```
while (1)
    printf("\n");
```

(xiii) The different storage classes in C language are static, auto, register, extern.

OR

When a function calls itself, the process is called recursion.

(xiv) a++ means post increment *i.e.*, the variable 'a' is increased by 1 after an operation.

++a means pre increment *i.e.*, the variable 'a' is increased by 1 before an operation.

GROUP – C

3. Answer the following questions : (Alternatives are to be noted)

(i) (a) Two functions of ALU are :

1. Based on the control signals, the ALU carries out various arithmatic operations like addition, subtraction, multiplication, or division.
2. It also carries out various logical operations on data like AND, OR, NOT operations.

(b) Short Note on Laser Printer :

Laser printers are non-impact printers that use dry ink for printing. It can print both text and images on paper. During the printing process a laser beam is used to charge portions of a roller drum coated with a special photo-conducting material. As paper is rolled over the drum, dry ink gets deposited on the paper based on the static charge on the drum. The ink is then bonded onto the paper using heat.

Both black and white and colour printing can be done using a laser printer.

(c) Short Note on Barcode Reader :

A barcode reader is an input device that is used to scan product information from product labels. The information on a barcode usually contains the product ID, price, name, manufacturers etc.

A standard barcode consists of a number of parallel light and dark lines / bars of variable width. The code is read optically using a special scanner. Two types of barcode systems available are UPC (Uniform Product Code) and EAN (European Article Number).

OR

(a) 3 characteristics of 2nd generation of computers are :

1. These computers used transistors
2. These computers produced much less heat compared to first generation of computes.
3. These computers were faster than first generation computers.

(b)

Difference	
Data	Information
1. Represents the raw material on which the computer works.	1. Represents the processed raw material.
2. It is the starting material of data processing.	2. It is the end product of data processing.

(c) Short Note on DVD :

DVD stands for Digital Versatile Disk or Digital Video Disk. It is an optical secondary storage medium. DVD's are available as single layer with 4.7 GB capacity and dual layer DVD with 8.54 GB capacity. Dimension wise a DVD is 12 cm in diameter. It uses a 650 nanometer laser ray to read data from a DVD. Data is written on a DVD using the same technique of land and pit as used in a CD. It is available as both read-only and read-write form.

(ii) (a)
$$\begin{array}{r} {}^{B}C\ 12\ D\ F_{16} \\ -A\ B\ 6\ E_{16} \\ \hline 1\ 7\ 7\ 1 \end{array}$$

Ans : $(-1771)_{16}$

(b) $1\ 2\ A\ 7\ .\ 1\ 3_{16}$

$0001\ 0010\ 1010\ 0111\ .\ 0001\ 0011$

Ans : $(1001010100111.00010011)_2$

(c)
$$\begin{array}{r} 1\ 1\ 1 \\ 1\ 0\ 1\ 1_2 \\ +\ 1\ 1\ 0\ 1_2 \\ \hline 1\ 1\ 0\ 0\ 0 \end{array}$$

Ans : $(11000)_2$

OR (a) $1101 - 1001$

$= 1101 + (1\text{'s comp. of } 1001)$

$= 1101 + 0110$

Ans : $(100)_2$

$$\begin{array}{r} 1 \\ 1101 \\ +\ 0110 \\ \hline (1)0011 \\ \quad \rightarrow +1 \\ \hline 100 \end{array}$$

(b) The ASCII code is used for representing characters in a computer system. The full form of ASCII is American Standard Code for Information Interchange.

It uses 7 bits to encode each character.

For example :

ASCII values 65 to 90 represent the capital alphabets A to Z and ASCII values 97 to 122 represent the lower case alphabets a to z.

(c) The Gray code of $(1101)_2$ is :

$g_3 \quad g_2 \quad g_1 \quad g_0$

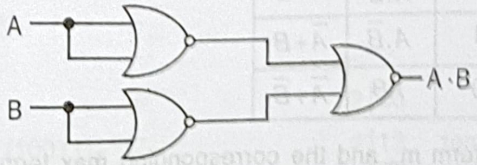
$1, \quad 1 \oplus 1, \quad 1 \oplus 0, \quad 0 \oplus 1 \quad i.e. \quad (1011)_2$

(iii) (a) Any logic gate can be formed using a NOR gate. Hence a NOR gate is called an universal gate.

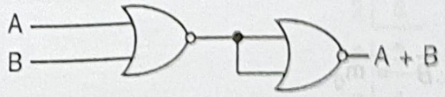
NOT gate :



AND gate :

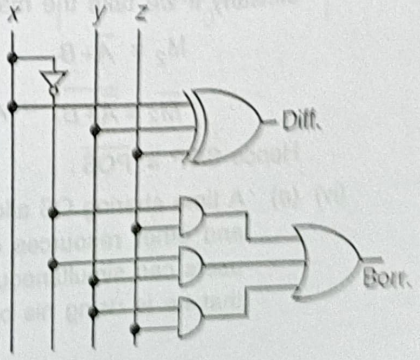


OR gate :



(b) Truth table of full subtractor :

x	y	z	Diff.	Borr.
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1



(c) Simplify : $F = (A+\bar{B}+C)(\bar{A}+\bar{B}+C)(A+\bar{B}+\bar{C})(\bar{A}+\bar{B}+\bar{C})$

$= (A.\bar{A}+\bar{B}+C).(A.\bar{A}+\bar{B}+\bar{C})$ $\quad [(X + Y)(X + Z) = X + YZ]$

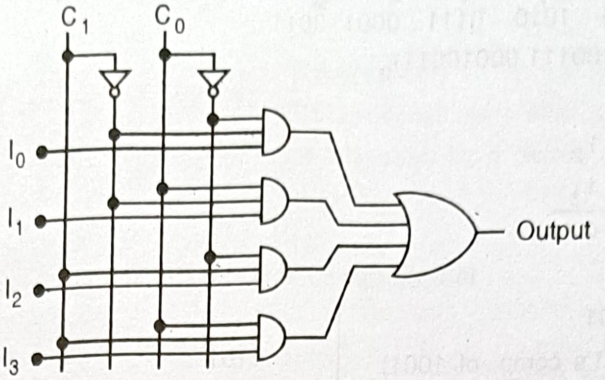
$= (\bar{B}+C)(\bar{B}+\bar{C})$ $\quad [As\ (X + Y)(X + Z) = X + YZ]$

$= \bar{B}+C.\bar{C} \quad [C.\bar{C}=0]$

$= \bar{B}$ Ans.

(iii) OR

(a) 4 : 1 MUX



(b) $F = (A+\bar{B}).(\bar{A}+B)$

$\therefore \bar{F} = \overline{(A+\bar{B}).(\bar{A}+B)}$

$= \overline{A\bar{A}+AB+\bar{A}\bar{B}+\bar{B}B}$

$= \overline{AB+\bar{A}\bar{B}}$ [As. $X.\bar{X}=0$]

$= (\overline{AB}).(\overline{\bar{A}\bar{B}})$ [Using de Morgan's Theorem]

$= (\bar{A}+\bar{B}).(A+B)$

$= \bar{A}A+\bar{A}B+A\bar{B}+\bar{B}B$

$= \bar{A}B+A\bar{B}$ [As $X.\bar{X} = 0$] Ans.

(c) Let us consider the following truth table

A	B	f	min	max
0	0	0	$\bar{A}.\bar{B}$	$A + B$
0	1	1	$\bar{A}.B$	$A+\bar{B}$
1	0	1	$A.\bar{B}$	$\bar{A}+B$
1	1	0	AB	$\bar{A}+\bar{B}$

If we take the min term m_0 and the corresponding max term M_0. Then :

$M_0 = A + B$

$\therefore \quad \bar{M}_0 = \overline{A+B} = \bar{A}.\bar{B} = m_0$

Similarly if we take the max term M_2 then :

$M_2 = \bar{A}+B$

$\therefore \quad \bar{M}_2 = \overline{\bar{A}+B} = A.\bar{B} = m_2$

Hence SOP = $\overline{POS}$.

(iv) (a) A time sharing OS allows various users to share the total available time of the CPU memory and other resources of the computer system. Time sharing is the process by which many users can simultaneously use a computer system such that each user is given the impression that he is using his own computer.

Examples of such systems are VAX / VMX and UNIX workstations.

(b) Virtual memory is a memory management scheme that allows the execution of process even in case of insufficient free main memory. To do this a portion of the memory of a high speed secondary storage device like a hard disk is used along with the main memory. The operating system uses this secondary memory as an additional memory area to the main memory.

(c) The attrib command in DOS is used to set the attribute for a file. The options available with ATTRIB are :

+*a* for archive file

+*r* for read file

+*s* for system file

+*h* for hidden file

e.g. c : \ > ATTRIB +*r* +*a* mydata.txt

(iv) OR

(a) A batch file is used to group several commands into a file with the extension .bat. When the batch file is executed, the commands in the batch file are executed one by one without any human intervention. The EDIT command in MS DOS can be used to create a Batch file.

(b) A CPU may have multiple independent processors that work in parallel. Such computer systems are required for advanced scientific research and commercial application which require a great deal of processing power. An operating system that supports such a hardware configuration and helps to share the computer resources is called a multiprocessing operating system.

Example includes VAX / VMS and UNIX workstations.

(c) Wildcard characters make it easy to deal with a set of files and directories with a single command. It also helps one to search for files even if the file name is not known exactly.

UNIX uses three types of wild card characters. These are :

1. * is used to represent any number of characters
2. ? is used to match a single character
3. [ ] is used to match specific character

(v) (a) In an entry controlled loop the loop condition is checked at the beginning of the loop. A while loop is an example of an entry controlled loop.

```
e.g. int i = 1;
     while ( i < = 10 )
         { printf ("\n %d", i );
           i++;
         }
```

(b) 25_{10} & 19_{10}

$= (11001)_2$ & $(10011)_2$

```
= 1 1 0 0 1
  1 0 0 1 1
  ---------
  1 0 0 0 1
```

$= (10001)_2$ Ans. [Out of syllabus]

2 | 25		2 | 19	
2 | 12	rem 1	2 | 9	rem 1
2 | 6	0	2 | 4	1
2 | 3	0	2 | 2	0
2 | 1	1	2 | 1	0
0	1	0	1

(c)
```
# include <stdio-h)
void main ( )
    { int num, digit, rev = 0, copy;
    printf ("\n Enter number :");
    scanf ("%d", &num);
    copy = num;
```

```
    while (copy > 0)
        {digit = copy%10;
        rev = rev * 10 + digit;
        copy = copy/10;
        }
    if (num = = rev)
        printf ("\n palindrome number");
    else
        printf ("\n Not a palindrome number")
    }
```

(v) OR

(a) Nested if statement :

```
if (x >= 65)
    { if (x <= 90)
        printf ("Uppercase")
      else
        printf ("Not uppercase")
    }
```

```
if (condition)
    { if (condition 2)
        Statement;
      else
        Statement;
    }
```

(b) $(21)_{10} \mid (19)_{10}$

$= (10101)_2 \mid (10011)_2$

$= 1\ 0\ 1\ 0\ 1$

$\quad 1\ 0\ 0\ 1\ 1$

$\quad 1\ 0\ 1\ 1\ 1$

2 | 21		
2 | 10	rem	1
2 | 5		0
2 | 2		1
2 | 1		0
0		1

2 | 19		
2 | 9	rem	1
2 | 4		1
2 | 2		0
2 | 1		0
0		1

Ans. $(10111)_2$ [Out of syllabus]

(c)
```
#include <stdio.h>
void main ( )
    { int i, j, k;
      for (i = 5; i >= 1; i – –)
        { for (j = 1, j <= 5 – i; j++)
            printf (" ␣ ");
          for (k = 1; k <= i; k++)
            printf ("*");
          printf ("\n");
        }
    }
```

[Note : ␣ represents a blank space].

QUESTIONS OF ANNUAL EXAMINATION, 2018
COMPUTER SCIENCE
CLASS-XI
(PRACTICAL)
NEW SYLLABUS

[Time : 3 hours] [Full Marks : 30]

Special credit will be given for answers which are brief and to the point.
Marks will be deducted for spelling mistakes, untidiness and bad handwriting.
Figures in the margin indicate full marks for the questions.

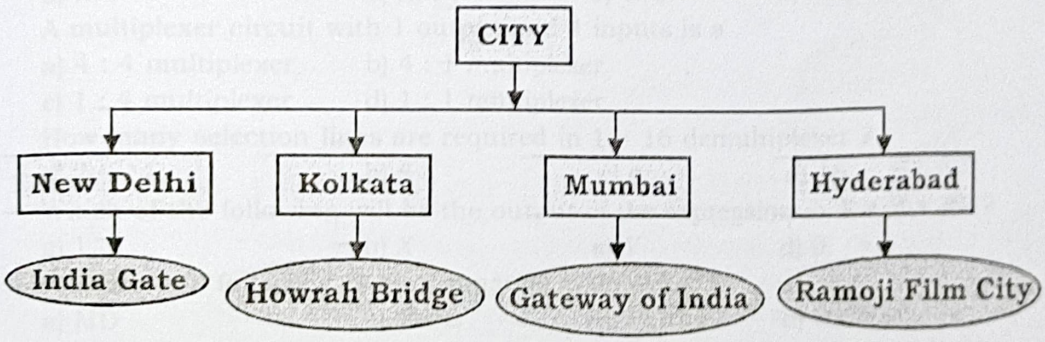
General Instructions :

- There are 4 (four) groups. Group-A contains 4 (four) questions. Group-B contains 2 (two) parts, each containing 4 (four) questions. You have to answer 1 (one) question from Group-A and 1 (one) question from each part of Group-B.
- Write all the steps in your answer-script which you have performed with the computer.
- Print all files, if necessary and possible, otherwise write all files with partial data input and output in your answer-script.
- Make suitable assumptions, if any, and tabulate them.

GROUP-A

Answer any *one* from the following questions : $5 \times 1 = 5$

1. Carry out the following using the Windows operating system :
   a) Rename any one existing folder of Desktop.
   b) Convert all files and folders into 'Tiles' view of any one drive.
   c) Change the Desktop Background.
   d) Find out the files having '.txt' file extension name of any drive. 1+1+1+2
2. a) Create a folder 'Games' on the Desktop. Inside this folder create three more folders namely 'Cricket', 'Football' and 'Tennis'.
   b) Delete the 'Football' folder and again retrieve the folder at its original position. 3 + 2
3. Using Unix/Linux operating system commands, create the following tree : 5



Here boxes are denoting directories and oval-shapes are files.

4. a) Show the list of all current users using Unix/Linux command.
   b) Create a file named 'INDIA' which will contain "I love my country" using suitable command in Unix/Linux.
   c) By using Unix/Linux command, show the present working directory and system date. 1 + 2 + 2

GROUP-B

PART-I

Answer any *one* from the following questions : $5 \times 1 = 5$

- Algorithm / Flow chart : 2
- Program coding in 'C' language : 2

- Execution and output : 1

1. Write a program in 'C' language to check an integer number whether it is even or odd. The number should be given as input except zero. 5
2. Write a 'C' program to take two strings as input and concatenate them. 5
3. Write a program in 'C' language to check an integer number whether it is prime or not. The number will be taken as input. 5
4. Write a program in 'C' language to make the following pattern :

```
* * * *
* * *
* *
*
```

5

PART-II

Answer any *one* from the following questions : $10 \times 1 = 10$

- Algorithm / Flow chart : 4
- Program coding in 'C' language : 4
- Execution and results : 2

1. Write a program in 'C' language, which will take an array of 5 numbers and find out the largest and smallest numbers of that array. 10
2. Write a 'C' program which can print the following series :
   0, 1, 1, 2, 3, 5, ..., n (Fibonacci series)
   Value of n will be inputted from user. 10
3. Write a program in 'C' language to convert all the lower case letters in a sentence to the corresponding upper case letters. 10
4. Write a program in 'C' language to add two 3 × 3 matrices and store into another matrix and as well as display. 10

GROUP-C

Laboratory notebook 5

GROUP-D

Viva-voce 5

QUESTIONS OF ANNUAL EXAMINATION, 2018
COMPUTER SCIENCE
CLASS-XI
NEW SYLLABUS

[Time : 3 Hours 15 minutes] [Full Marks : 70]

Special credit will be given for answers which are brief and to the point.
Marks will be deducted for spelling mistakes, untidiness and bad handwriting.
Figures in the margin indicate full marks for the questions.

GROUP-A

1. Answer the following questions : $1 \times 21 = 21$

i) The technology of second generation computer is
a) Transistor b) IC c) VLSI d) LSI.

ii) Magnetic tape is an ..... device.
a) Input b) Output c) Processing d) Memory

iii) $(32)_8 + (56)_8 = (?)_8$
a) 121 b) 110 c) 88 d) 1001000.

iv) ASCII value of A is
a) 97 b) 35 c) 65 d) 90.

v) The value of $A + A = ?$ (Using Boolean Algebra)
a) $2A$ b) 1 c) 0 d) A.

vi) The number of minterms for n variables is
a) n b) n^2 c) 2^n d) None of these.

vii) SOP form is one of the
a) Boolean expression b) Maxterms
c) Minterms d) Canonical expression.

viii) The dual form of the Boolean expression $A + \bar{C}$ is
a) $\bar{A}.\bar{C}$ b) $A.\bar{C}$ c) $\bar{A}.C$ d) $A.C$

ix) A multiplexer circuit with 1 output and 4 inputs is a
a) 4 : 4 multiplexer b) 4 : 1 multiplexer
c) 1 : 4 multiplexer d) 1 : 1 multiplexer.

x) How many selection lines are required in 1 × 16 demultiplexer ?
a) 2 b) 8 c) 4 d) 16

xi) Which of the following will be the output of the expression → $F = X + XY$?
a) 1 b) X c) Y d) 0.

xii) which of the following DOS commands is External ?
a) MD b) TYPE c) EDIT d) DATE.

xiii) The UNIX command used to rename a file is
a) rd b) cp c) mv d) ls.

xiv) Which of the following is *not* a system defined header file in C-language ?
a) stdio.h b) conio.h c) math.h d) function.h

xv) Which of the following functions display the whole string on the screen in C-language ?
a) gets( ) b) puts( ) c) scanf( ) d) None of these.

xvi) What type of operator is '&&' in C-language ?
a) Logical operator b) Relational operator
c) Arithmetic operator d) Conditional operator.

xvii) The memory size of a signed integer will be ..... bit.
a) 16 b) 8 c) 4 d) None of these.

xviii) To take a 5 field width integer type data as input, what will be the syntax ?
a) scanf ("%d", & 5a); b) scanf ("5% d", & a);
c) scanf ("%5 d", & a) d) scanf ("% d5", & a).

xix) ..... stores the address of a variable, but not value of another variable.
a) Array b) String c) Pointer d) Function.

xx)
```
# include < stdio.h >.
void main ( )
{
    int x = 4, y, z ;
    y = - -x ;
    z = x - - ;
    printf ( "% d % d % d", x, y, z ) ;
}
```
Here the output will be
a) 323 b) 223 c) 322 d) 233.

xxi)
```
# include < stdio.h >
void main ( )
{
int x = 4 ;
int * p = & x ;
int * k = p ** ;
int r = p - k ;
printf("% d,"r);
}
```
Here the output will be
a) 4 b) 8 c) 1 d) Runtime error.

GROUP-B

2. Answer the following questions in brief (Alternatives are to be noted) : $1 \times 14 = 14$

i) Write two disadvantages of super computer.

OR

4 GB = ? bytes.

ii) $(A702)_{16} = (?)_8$

OR

$(10100)_2 - (1111)_2 = (?)_2$.

iii) Write the truth table of 2-input XNOR gate ?

OR

Using truth table prove that $\overline{A+B} = \bar{A} \cdot \bar{B}$.

iv) What do you mean by Multitasking operating system ?

v) Define variable.

vi) In *C*-language what is the significace of 'break' statement?

OR

What is the function of 'go-to' statement in *C*-language ?

vii) What is the difference between '=' and '= =' in *C*-language ?

viii) What do you mean by exit controlled loop ?

OR

Write the syntax of 'do-while' loop in *C*-language.

ix) If $x = 6$ and $y = 1$ then, what will be the value of $x >> y$ = ?

OR

Write the syntax of 'switch case' in *C*-language.

x) Why do we use < conio.h > in a *C*-program ?

xi) What is function prototype ?

OR

What is recursion ?

xii) Define array.

OR

Write the syntax to define a structure in *C*-language.

xiii) Let $a = 5$ and $b = 2$. Write the *C* code to swap the values of these variables without using third variable.

xiv)
```
# include < stdio.h >
int main ( )
{
        int counter ;
        for (Counter = 20; Counter > = 1; Counter – = 2)
         {
        printf ("\n% d", counter) ;
         }
        return 0 ;
}
```

What will be the output of the above *C*-code ?

OR

What is Post increment ?

GROUP-C

3. Answer the following questions (Alternatives are to be noted) : $7 \times 5 = 35$

i) a) Discuss Cache memory in short.

b) Write two characteristics of 5th Generation computer.

c) Write a short note on OCR. 3 + 2 + 2

Or

a) Discuss Flash memory in short.

b) Write two characteristics of Mainframe Computer.

c) Write a short note on MICR. 3 + 2 + 2

ii) a) Calculate the value of x :

$$(x)_{16} + (10111)_2 = (167)_8$$

b) Find out the BCD code of $(23)_{10}$.

c) Write full form of ISCII and EBCDIC. 3 + 2 + 2

OR

a) Subtract by using 2's complement method :

$$(110100)_2 - (10111)_2 = (?)_2$$

b) $(A6D)_{16} + (12E)_{16} = (?)_{16}$.

c) What is Signed Magnitude Number ? Give example. 3 + 2 + 2

iii) a) Write the truth table of a full adder and find out the logic expressions of it using *K*-map and also draw the logic diagram.

b) $f(A, B, C) = \Sigma(0, 2, 4, 7)$, write down the truth table and find out the logic expression. (1 + 3) + 3

OR

a) Write down the truth table for the following logic function. Simplify it as a product of sums form using Karnaugh map.

$f(A, B, C) = \pi(1, 3, 5, 6, 7)$.

b) Draw the circuit diagram of Full adder by using Half Adder.

c) What is the purpose of enable input in a decoder ? (2 + 2) + 2 + 1

iv) a) What is system software ?

b) Define compiler.

c) Write two differences between High level language and Low lavel language.

d) What is a Mnemonic ? 2 + 2+ 2 + 1

OR

a) Write down the function and syntax of following UNIX commands :
CP and *Cat*.

b) Write down the function and syntax of the DIR command in MS-DOS.

c) What does an assembler do ? (2 + 2) + 2 + 1

v) a) Write the *C* program to make the following structure (The number of lines will be taken as input) :

```
1
1  3
1  3  5
1  3  5  7
```

b) Write the differences between 'while' and 'do-while' loop in *C*-language.

c) What is the function of Putchar( ) ? 4 + 2 + 1

OR

a) Write a progarm in *C*-language to check a number is Armstrong or not.

b) Differentiate between Local variable and Global variable.

c) What is the function of gets ( ) ? 4 + 2 + 1

QUESTIONS OF ANNUAL EXAMINATION, 2019
COMPUTER SCIENCE
CLASS-XI
(PRACTICAL)
NEW SYLLABUS

Time : 3 hours] [Full Marks : 30]

Special credit will be given for answers which are brief and to the point.
Marks will be deducted for spelling mistakes, untidiness and bad handwriting.
Figures in the margin indicate full marks for the questions.

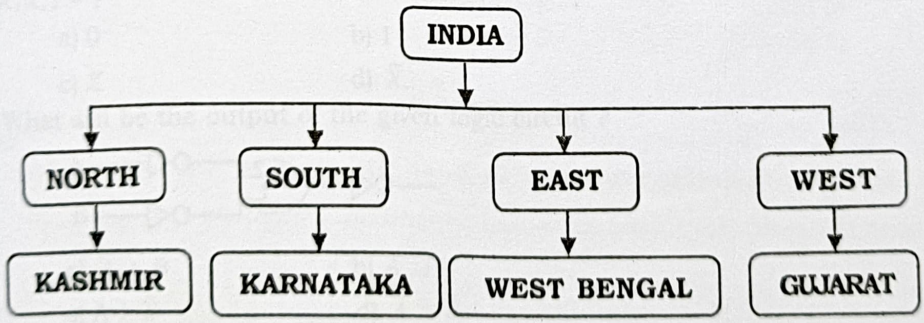
General Instructions :

- There are 4 (four) groups. Group-A contains 4 (four) questions. Group-B contains 2 (two) parts, each containing 4 (four) questions. You have to answer 1 (one) question from Group-A and 1 (one) question from each part of Group-B.
- Write all the steps in your answer-script which you have performed with the computer.
- Print all files, if necessary and possible, otherwise write all files with partial data input and output in your answer-script.
- Make suitable assumptions, if any, and tabulate them.

GROUP-A

Answer any *one* from the following questions : $5 \times 1 = 5$

1. Carry out the following using the Windows operating system :
   a) Find all files with the file extension .rtf in your computer.
   b) Create a folder of Desktop and Rename it as 'COMS PRACTICAL EXAM'.
   c) Sort the files in C drive by name. 2 + 2 + 1
2. a) Create a.bmp file using the MS-Paint program and then save it on the Desktop as .gif file.
   b) Arrange all icons on Desktop according to their type.
   c) Create an arbitrary file in your Desktop. Send the file to Recycle Bin. Finally retrieve the file from Recycle Bin. 2 + 1 + 2
3. Using Unix/Linux operating system commands, create the following tree : 5



Here boxes are denoting directories and circles are files.

4. a) Using Unix/Linux command, display all current users of system.
   b) Using Unix/Linux, create a file 'Test' containing the sentence "This is a line".
   c) Make the file 'Test' hidden. Enable read and write permission of the file 'Test' for all users. 1 + 2 + 2

GROUP-B

PART-I

Answer any *one* from the following questions : $5 \times 1 = 5$

- Algorithm/Flow chart : 2
- Program coding in 'C' language : 2
- Execution and output : 1

1. Write a *C* program that checks whether an input alphabet is a vowel or consonant. (Check both lower case and upper case) 5
2. Write a program in *C* language to find out the summation of all odd numbers in between 1 to *n*, where *n* is a positive integer. After execution, show output. 5
3. Write a program in *C* language to print digits of an inputted number in reverse. 5
4. Write a *C* program to generate the following pattren :

```
1
1 3
1 3 5
1 3 5 7
1 3 5 7 9
```

PART-II

Answer any *one* from the following questions : $10 \times 1 = 10$

- Algorithm/Flow Chart : 4
- Program coding in 'C' language : 4
- Execution and results : 2

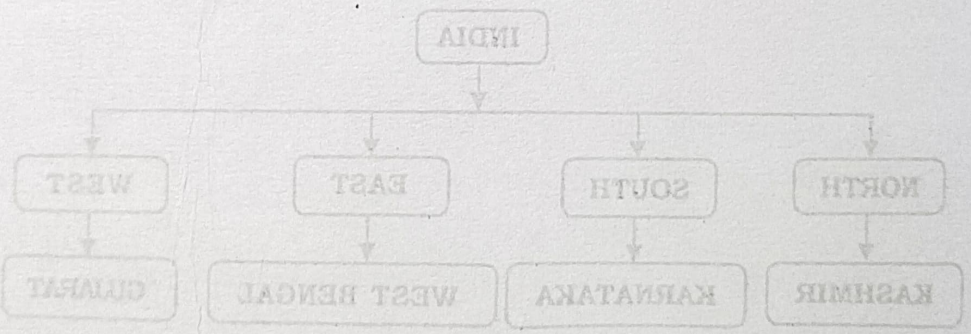
1. Write a *C* program to find out the sum of *n* natural numbers using recursion, where n is a positive integer. 10
2. Write a *C* program to check whether an inputted string is Palindrome or not without using strrev() function. Display appropriate message as output. 10
3. Write a *C* language program to generate Armstrong Number.
   [A number is Armstrong if the sum of cubes of individual digits of a number is equal to the number itself. e.g. $371 = 3^3 + 7^3 + 1^3$] 10
4. Write a program to generate a Transpose Matrix of a 3 × 3 matrix using *C* language. 10

Group-C

Laboratory notebook

Group-D

Viva-voce



QUESTIONS OF ANNUAL EXAMINATION, 2019
COMPUTER SCIENCE
CLASS-XI
NEW SYLLABUS

[Time : 3 hours 15 minutes] [Full Marks : 70]

Special credit will be given for answers which are brief and to the point.
Marks will be deducted for spelling mistakes, untidiness and bad handwriting.
Figures in the margin indicate full marks for the questions.

GROUP-A

1. Answer the following questions : $1 \times 21 = 21$

i) AI is used in which generation of computer ?
a) First b) Third
c) Fourth d) Fifth.

ii) Which is not a impact printer ?
a) Inkjet b) Dot matrix
c) Line d) Drum.

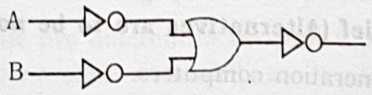
iii) What is the Gray code of $(101)_2$?
a) $(101)_2$ b) $(111)_2$
c) $(010)_2$ d) $(110)_2$

iv) 10's complement of $(232)_{10}$ is
a) 786 b) 768
c) 678 d) 767.

v) What is the dual of Boolean expression $A + \bar{B}C$?
a) $A.\ \bar{B}C$ b) $A\ (\bar{B} + C)$
c) $\bar{A}(B + \bar{C})$ d) $\bar{A}.\ B\bar{C}$.

vi) $X.\bar{X}.1 = ?$
a) 0 b) 1
c) X d) $\bar{X}$.

vii) What will be the output of the given logic circuit ?



a) $\bar{A} + B$ b) $A\ .\ B$
c) $A + \bar{B}$ d) $\bar{A} + \bar{B}$.

viii) How many inputs are there in a decimal to binary encoder ?
a) 2 b) 8
c) 10 d) 16.

ix) The Σ notation of SOP expression $F(A, B, C) = \bar{A}BC + A\bar{B}C + AB\bar{C}$ is
a) Σ(3, 4, 5) b) Σ(1, 2, 4)
c) Σ(3, 4, 6). d) Σ(2, 5, 7)

x) There are two XOR gates, two AND gates and one ..... in a full Adder circuit.
a) NOR gate b) OR gate
c) NOT gate d) NAND gate.

xi) In combinational circuit data cannot be stored because there is no ..... here.
a) OR gate b) NAND gate
c) Memory d) None of these.

xii) Which is Not an Application software ?
a) Foxpro b) Assembler
c) Games d) Spreadsheet.

xiii) The text editor of Unix OS is
a) ed b) vi
c) edit d) notepad.

xiv) How many relational operators are there in *C* ?
a) 3 b) 4
c) 6 d) 8.

xv) What will be the output of the following *C*-code ?
int $x = 2$;
printf ("%d %d %d", x, x^{++}, ^{++}x) ;
a) 4 4 3 b) 2 2 2
c) 2 2 4 d) 4 3 3

xvi) The library function to add two strings in *C* are
a) strcmp ( ) b) Strlen ( )
c) Strstr ( ) d) Strcat ( ).

xvii) Which one of the following stores the address of the variable not value ?
a) Array b) String
c) Pointer d) Function.

xviii) Which header file is included for clrscr ( ) and getch ( ) functions ?
a) math.h b) Stdio.h
c) stdlib.h d) conio.h

xix) int i ;
for ($i = 0$; $i <= 4$;)
printf ("%d", $^{++} i$) — the output is
a) 0123 b) 12345
c) 1234 d) 123.

xx) Char X [ ] = "A" ; Which statement is correct for this array ?
a) X [0] = "A" b) X = "ABC"
c) X [1] = "A" d) X = "A"

xxi) ..... is used to take user input through keyboard in *C*.
a) printf b) scanf
c) print d) scan.

GROUP-B

1. Answer the following questions in brief (Alternatives are to be noted) : $1 \times 14 = 14$

i) Name the technology used in second generation computers.

OR

What is the use of Drag and Drop in mouse ?

ii) $(10.01)_2 \times (1.01)_2 = (?)_2$

OR

$(-10110)_2$ — express it in 1's complement form of 8 bit.

iii) Prove that $\overline{A+B} = \overline{A} \cdot \overline{B}$.

OR

Write the truth table of Half subtractor.

iv) What do you mean by Time Sharing operating system ?

OR

What are the wild card characters of DOS ?

v) Write two characteristics of flowchart ?

OR

What is algorithm ?

vi) What is the range of value storing of int data type in *C* ?

OR

Write the statement for printing 1 to 10 numbers ?

vii) What is the difference between '*a*' and "*a*" in *C* ?

viii) What will happen if there is no stopping condition in recursive function ?

OR

Write the difference between Normal function and Recursive function in *C* ?

ix) What will it mean if strlen ( ) function returns zero value in *C* ?

OR

Write down the correct statement ?

i) Char *var* = "*A*" ;

ii) float arr *a* { 15 };

x) If *a* = 30, *b* = 40, then write the *C*-code to interchange the values of a and b without using a third variable.

xi) What will be the output of ("%5.3 *f*" , 125.11) ?

OR

What do you mean by call by reference ?

xii) Give an example of array of structure ?

OR

Write one difference between while and do-while loop.

xiii)
```
int x = 1, y = 1 ;
if ( n > 0 )
{
    x = x + 1;
    y = y – 1 ;
}
    printf ( "%d %d" , x, y ) ;
```

What will be the values of *x*, *y* if *n* = 1 ?

xiv) Which signs or operators are used for using pointer variables ?

OR

If *a* = 20 and *P* = & *a*, then what will be the value od * *p* + 1 ?

GROUP-C

❑ **3. Answer the following questions (Alternatives are to be noted) :** 7 × 5 = 35

(i) (a) Write two differences between Calculator and Computer.

(b) Write a short note on Laser Printer.

(c) Write the differences between SRAM and DRAM. 2 + 2 + 3

OR

(a) Write down two characteristics of Third Generation computers.

(b) Write the differences between internal bus and external bus.

(c) There are 20 disc plates in a hard disk. There are 100 tracks in each plate, 25 sectors in each track and 5 kB of data in each sector. What is the capacity of the hard disk ? 2 + 2 + 3

(ii) (a) $(563)_8 + (486)_8 = (?)_8$

(b) $(BCE)_{16} + (17D)_{16} = (?)_{16}$

(c) Subtract : $(111111100)_2 - (537)_8$ 2 + 2 + 3

OR

(a) Subtract using 2's complement :
$(1011.01)_2 - (101.01)_2$

(b) What is the equivalent BCD of $(754)_{10}$?

(c) Write the differences between ASCII and EBCDIC ? 3 + 2 + 2

(iii) (a) $Y = \bar{A}B + A\bar{B}$

(i) Draw the truth table of the given boolean expression.

(ii) Draw the logic diagram using only NAND gate of the given boolean expression.

(b) Write two functions of Encoder.

(c) Draw the truth table of the 4 × 1 MUX (1 + 2) + 2 + 2

OR

(a) Draw the truth table and evaluate the logic expression of $F(A, B, C) = \Sigma(0, 2, 4, 7)$.

(b) Simplify using K-Map : $F(A, B, C) = \bar{A}BC + A\bar{B}C + ABC$. 4 + 3

(iv) (a) Write the differences between System software and Application software.

(b) What is Compiler ? Write the names of two languages where compiler is used.

(c) What is Mnemonics ? 2 + (2 + 2) + 1

OR

(a) Write a short note on Loader.

(b) What is the function of Virtual memory ?

(c) What is the function of *ls* and *Cp* command in Unix (with example) ? $2 + 2 + (1\frac{1}{2} + 1\frac{1}{2})$

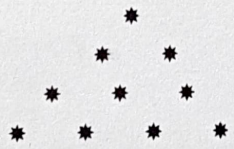
(v) (a) Write a *C* program to print fibonacci series (using Recursive function).

(b) Write the differences between 'break' and 'continue' statement in *C*.

(c) Write one advantage of using flowchart. 4 + 2 + 1

OR

(a) Write a *C* program to print the following pattern.



(b) Draw a flowchart to examine whether a given number is odd or even 4 + 3

About the book...

The book has been written strictly according to the newly revised bifurcated syllabus for "Computer Science" as prescribed by the West Bengal Council for Higher Secondary Education for classes 11 and 12

The book has been written in a lucid, easy to understand language suitable for the understanding of students in the plus 2 level

Various formatting and pictorial techniques used throughout the book to make the searching of important topics an easy process

Extensive exercises provided at the end of each chapter to make the learning experience more fruitful

More than 150 worked-out programs done using C and the working of each program explained with the help of various diagrams and possible outputs

Common logical and syntactical errors while writing a program have been treated and illustrated separately at the end of each chapter on C

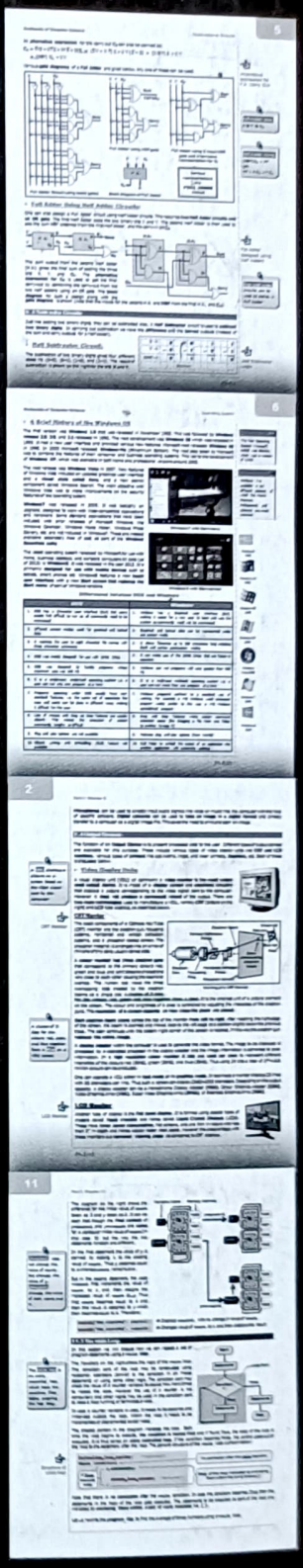
Chapter summary given at the end of each chapter to help in recapitulation

General index provided at the end of the book to help find any topic easily

Various useful appendixes provided at the end of the book that include:

a. The ASCII character set
b. List of DOS and UNIX commands
c. List of C commands
d. A Program List that covers all the worked-out programs in the book
e. List of various Acronyms used in the book
f. List of tables dealing with various differences and comparisons

A set of theory and practical examination question papers of the previous years of WBCHSE class 11 examinations provided at the end of the book



About the author...



Joyrup Bhattacharya is head of the department of South Point High School, Computer Department, for more than sixteen years. His innovative methods in teaching the subject has been well appreciated by his students. Apart from teaching he also takes interest in photography, painting, and Indian classical music.

ISBN 9789383420643

Books available at ACADEMIC PUBLISHERS www.academicpublishers.in flipkart Google play